AD-A234 439

ENTATION PAGE

Form Approved
OPM No. 0704-0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
		Final: Nov 30, 1990 to Mar 1, 1993

4. TITLE AND SUBTITLE
Ada Compiler Validation Summary Report: DDC International A/S, DACS 80386 UNIX V Ada Compiler System, Version 4.6, ICL DRS300 (Host & Target), 901129S1.11075

5. FUNDING NUMBERS

6. AUTHOR(S)
National Institute of Standards and Technology
Gaithersburg, MD
USA

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
National Institute of Standards and Technology
National Computer Systems Laboratory
Bldg. 255, Rm A266
Gaithersburg, MD 20899 USA

8. PERFORMING ORGANIZATION REPORT NUMBER
NIST90DDC500_4_1.11

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Ada Joint Program Office
United States Department of Defense
Pentagon, RM 3E114
Washington, D.C. 20301-3081

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
DDC International A/S, DACS 80386 UNIX V Ada Compiler System, Version 4.6, Gaithersburg, MD, ICL DRS300 running DRS/NX, Version 3.2 (UNIX System V/386 release 3.2)(Host & Target), ACVC 1.11.

14. SUBJECT TERMS
Ada programming language, Ada Compiler Val. Summary Report, Ada Compiler Val. Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANSI/MIL-STD-1815A, AJPO.

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFED	UNCLASSIFIED	

AVF Control Number: NIST90DDC500_4_1.11
DATE COMPLETED
BEFORE ON-SITE: October 30, 1990
AFTER ON-SITE: November 30, 1990
REVISIONS:

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 901129S1.11075
DDC International A/S
DACS 80386 UNIX V Ada Compiler System, Version 4.6
ICL DRS300 => ICL DRS300

Prepared By:
Software Standards Validation Group
National Computer Systems Laboratory
National Institute of Standards and Technology
Building 225, Room A266
Gaithersburg, Maryland 20899

AVF Control Number: NIST90DDC500_4_1.11

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on November 29,1990.

Compiler Name and Version:	DACS 80386 UNIX V Ada Compiler System, Version 4.6
Host Computer System:	ICL DRS300 running DRS/NX, Version 3.2 (UNIX System V/386 release 3.2)
Target Computer System:	ICL DRS300 running DRS/NX, Version 3.2 (UNIX System V/386 release 3.2)

A more detailed description of this Ada implementation is found in section 3.1 of this report.

As a result of this validation effort, Validation Certificate 901129S1.11075 is awarded to DDC International A/S. This certificate expires on March 01, 1993.

This report has been reviewed and is approved.

Ada Validation Facility
Dr. David K. Jefferson
Chief, Information Systems
Engineering Division (ISED)
National Computer Systems
Laboratory (NCSL)
National Institute of
Standards and Technology
Building 225, Room A266
Gaithersburg, MD 20899

Ada Validation Facility
Mr. L. Arnold Johnson
Manager, Software Standards
Validation Group
National Computer Systems
Laboratory (NCSL)
National Institute of
Standards and Technology
Building 225, Room A266
Gaithersburg, MD 20899

for

Ada Validation Organization
Director, Computer & Software
Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

Ada Joint Program Office
Dr. John Solomond
Director
Department of Defense
Washington DC 20301

DECLARATION OF CONFORMANCE

The following declaration of conformance was supplied by the customer.

DECLARATION OF CONFORMANCE

Customer and Certificate Awardee: DDC International A/S

Ada Validation Facility: National Institute of Standards and Technology
National Computer Systems Laboratory (NCSL)
Software Validation Group
Building 225, Room A266
Gaithersburg, Maryland 20899

ACVC Version: 1.11

Ada Implementation:

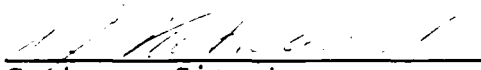
Compiler Name and Version: DACS 80386 UNIX V Ada Compiler System, Version 4.6

Host Computer System: ICL DRS300 running DRS/NX, Version 3.2 (UNIX System V/386 release 3.2)

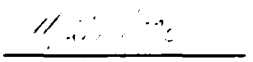
Target Computer System: ICL DRS300 running DRS/NX, Version 3.2 (UNIX System V/386 release 3.2)

Declaration:

[I/we] the undersigned, declare that [I/we] have no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A ISO 8652-1987 in the implementation listed above.



Customer Signature



Date

Company

Title

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro90] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro90]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.2 REFERENCES

[Ada83] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

[Pro90] Ada Compiler Validation Procedures, Version 2.1, Ada Joint Program Office, August 1990.

[UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued. Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values -- for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3.
For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing

withdrawn tests (see section 2.1) and, possibly some inapplicable tests (see Section 3.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, Validation consisting of the test suite, the support programs, the ACVC Capability user's guide and the template for the validation summary (ACVC) report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.
Conformity	Fulfillment by a product, process or service of all requirements specified.

Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro90].
Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

Some tests are withdrawn by the AVO from the ACVC because they do not conform to the Ada Standard. The following 81 tests had been withdrawn by the Ada Validation Organization (AVO) at the time of validation testing. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 90-10-12.

E28005C	B28006C	C34006D	B41308B	C43004A	C45114A
C45346A	C45612B	C45651A	C46022A	B49008A	A74006A
C74308A	B83022B	B83022H	B83025B	B83025D	B83026A
B83026B	C83041A	B85001L	C97116A	C98003B	BA2011A
CB7001A	CB7001B	CB7004A	CC1223A	BC1226A	CC1226B
BC3009B	BD1B02B	BD1B06A	AD1B08A	BD2A02A	CD2A21E
CD2A23E	CD2A32A	CD2A41A	CD2A41E	CD2A87A	CD2B15C
BD3006A	BD4008A	CD4022A	CD4022D	CD4024B	CD4024C
CD4024D	CD4031A	CD4051D	CD5111A	CD7004C	ED7005D
CD7005E	AD7006A	CD7006E	AD7201A	AD7201E	CD7204B
BD8002A	BD8004C	CD9005A	CD9005B	CDA201E	CE2107I
CE2117A	CE2117B	CE2119B	CE2205B	CE2405A	CE3111C
CE3118A	CE3411B	CE3412B	CE3607B	CE3607C	CE3607D
CE3812A	CE3814A	CE3902B			

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. The inapplicability criteria for some tests are explained in documents issued by ISO and the AJPO known as Ada Issues and commonly referenced in the format AI-dddd. For this implementation, the following tests were inapplicable for the reasons indicated; references to Ada Issues are included as appropriate.

The following 201 tests have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)
C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)

C45641L..Y (14 tests) C46012L..Z (15 tests)

C24113I..K (3 TESTS) USE A LINE LENGTH IN THE INPUT FILE WHICH EXCEEDS 126 CHARACTERS.

C35702A, C35713B, C45423B, B86001T, AND C86006H CHECK FOR THE PREDEFINED TYPE SHORT_FLOAT.

C35713D AND B86001Z CHECK FOR A PREDEFINED FLOATING-POINT TYPE WITH A NAME OTHER THAN FLOAT, LONG_FLOAT, OR SHORT_FLOAT.

C35404D, C45231D, B86001X, C86006E, AND CD7101G CHECK FOR A PREDEFINED INTEGER TYPE WITH A NAME OTHER THAN INTEGER, LONG_INTEGER, OR SHORT_INTEGER.

C45531M, C45531N, C45531O, C45531P, C45532M, C45532N, C45532O, AND C45532P CHECK FIXED-POINT OPERATIONS FOR TYPES THAT REQUIRE A SYSTEM.MAX_MANTISSA OF 47 OR GREATER.

C45624A CHECKS THAT THE PROPER EXCEPTION IS RAISED IF MACHINE_OVERFLOWS IS FALSE FOR FLOATING POINT TYPES WITH DIGITS 5. FOR THIS IMPLEMENTATION, MACHINE_OVERFLOWS IS TRUE.

C45624B CHECKS THAT THE PROPER EXCEPTION IS RAISED IF MACHINE_OVERFLOWS IS FALSE FOR FLOATING POINT TYPES WITH DIGITS 6. FOR THIS IMPLEMENTATION, MACHINE_OVERFLOWS IS TRUE.

C4A013B CONTAINS THE EVALUATION OF AN EXPRESSION INVOLVING 'MACHINE_RADIX APPLIED TO THE MOST PRECISE FLOATING-POINT TYPE. THIS EXPRESSION WOULD RAISE AN EXCEPTION. SINCE THE EXPRESSION MUST BE STATIC, IT IS REJECTED AT COMPILE TIME.

D56001B USES 65 LEVELS OF BLOCK NESTING WHICH EXCEEDS THE CAPACITY OF THE COMPILER.

C86001F RECOMPILES PACKAGE SYSTEM, MAKING PACKAGE TEXT_IO, AND HENCE PACKAGE REPORT, OBSOLETE. FOR THIS IMPLEMENTATION, THE PACKAGE TEXT_IO IS DEPENDENT UPON PACKAGE SYSTEM.

B86001Y CHECKS FOR A PREDEFINED FIXED-POINT TYPE OTHER THAN DURATION.

C96005B CHECKS FOR VALUES OF TYPE DURATION'BASE THAT ARE OUTSIDE THE RANGE OF DURATION. THERE ARE NO SUCH VALUES FOR THIS IMPLEMENTATION.

CA2009C, CA2009F, BC3204C, AND BC3205D THESE TESTS INSTANTIATE GENERIC UNITS BEFORE THEIR BODIES ARE COMPILED. THIS IMPLEMENTATION CREATES A DEPENDENCE ON GENERIC UNIT AS ALLOWED BY AI-00408 & AI-00530 SUCH THAT A THE COMPILATION OF THE GENERIC UNIT BODIES MAKES THE INSTANTIATING UNITS OBSOLETE.

CD1009C USES A REPRESENTATION CLAUSE SPECIFYING A NON-DEFAULT SIZE FOR A FLOATING-POINT TYPE.

CD2A84A, CD2A84E, CD2A84I..J (2 TESTS), AND CD2A84O USE REPRESENTATION CLAUSES SPECIFYING NON-DEFAULT SIZES FOR ACCESS TYPES.

THE TESTS LISTED IN THE FOLLOWING TABLE ARE NOT APPLICABLE BECAUSE THE GIVEN FILE OPERATIONS ARE SUPPORTED FOR THE GIVEN COMBINATION OF MODE AND FILE ACCESS METHOD.

Test	File Operation	Mode	File Access Method
CE2102E	CREATE	OUT_FILE	SEQUENTIAL_IO
CE2102F	CREATE	INOUT_FILE	DIRECT_IO
CE2102J	CREATE	OUT_FILE	DIRECT_IO
CE2102N	OPEN	IN_FILE	SEQUENTIAL_IO
CE2102O	RESET	IN_FILE	SEQUENTIAL_IO
CE2102P	OPEN	OUT_FILE	SEQUENTIAL_IO
CE2102Q	RESET	OUT_FILE	SEQUENTIAL_IO
CE2102R	OPEN	INOUT_FILE	DIRECT_IO
CE2102S	RESET	INOUT_FILE	DIRECT_IO
CE2102T	OPEN	IN_FILE	DIRECT_IO
CE2102U	RESET	IN_FILE	DIRECT_IO
CE2102V	OPEN	OUT_FILE	DIRECT_IO
CE2102W	RESET	OUT_FILE	DIRECT_IO
CE3102F	RESET	Any Mode	TEXT_IO
CE3102G	DELETE	--------	TEXT_IO
CE3102I	CREATE	OUT_FILE	TEXT_IO
CE3102J	OPEN	IN_FILE	TEXT_IO
CE3102K	OPEN	OUT_FILE	TEXT_IO
CE3109A	CREATE	IN_FILE	TEXT_IO

THE TESTS LISTED IN THE FOLLOWING TABLE ARE NOT APPLICABLE BECAUSE THE GIVEN FILE OPERATIONS ARE NOT SUPPORTED FOR THE GIVEN COMBINATION OF MODE AND FILE ACCESS METHOD.

Test	File Operation	Mode	File Access Method
CE2105A	CREATE	IN_FILE	SEQUENTIAL_IO
CE2105B	CREATE	IN_FILE	DIRECT_IO

CE2203A CHECKS FOR SEQUENTIAL_IO THAT WRITE RAISES USE_ERROR IF THE CAPACITY OF THE EXTERNAL FILE IS EXCEEDED. THIS IMPLEMENTATION CANNOT RESTRICT FILE CAPACITY.

EE2401D CHECKS WHETHER READ, WRITE, SET_INDEX, INDEX, SIZE, AND END_OF_FILE ARE SUPPORTED FOR DIRECT FILES FOR AN UNCONSTRAINED ARRAY TYPE. USE_ERROR WAS RAISED FOR DIRECT CREATE. THE MAXIMUM ELEMENT SIZE SUPPORTED FOR DIRECT_IO IS 32K BITS.

CE2403A CHECKS FOR DIRECT_IO THAT WRITE RAISES USE_ERROR IF THE CAPACITY OF THE EXTERNAL FILE IS EXCEEDED. THIS IMPLEMENTATION CANNOT RESTRICT FILE CAPACITY.

CE3111B AND CE3115A SIMULTANEOUSLY ASSOCIATE INPUT AND OUTPUT FILES WITH THE SAME EXTERNAL FILE, AND EXPECT THAT OUTPUT IS IMMEDIATELY WRITTEN TO THE EXTERNAL FILE AND AVAILABLE FOR READING; THIS IMPLEMENTATION BUFFERS FILES, AND EACH TEST'S ATTEMPT TO READ SUCH OUTPUT (AT LINES 87 & 101, RESPECTIVELY) RAISES END_ERROR.

CE3304A CHECKS THAT USE_ERROR IS RAISED IF A CALL TO SET_LINE_LENGTH OR SET_PAGE_LENGTH SPECIFIES A VALUE THAT IS INAPPROPRIATE FOR THE EXTERNAL FILE. THIS IMPLEMENTATION DOES NOT HAVE INAPPROPRIATE VALUES FOR EITHER LINE LENGTH OR PAGE LENGTH.

CE3413B CHECKS THAT PAGE RAISES LAYOUT_ERROR WHEN THE VALUE OF THE PAGE NUMBER EXCEEDS COUNT'LAST. THE VALUE OF COUNT'LAST IS GREATER THAN 150000 AND THE CHECKING OF THIS OBJECTIVE IS IMPRACTICAL.

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 64 tests.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

B22003A	B26001A	B26002A	B26005A	B28003A	B29001A	B33301B
B35101A	B37106A	B37301B	B37302A	B38003A	B38003B	B38009A
B38009B	B55A01A	B61001C	B61001F	B61001H	B61001I	B61001M
B61001R	B61001W	B67001H	B83A07A	B83A07B	B83A07C	B83E01C
B83E01D	B83E01E	B85001D	B85008D	B91001A	B91002A	B91002B
B91002C	B91002D	B91002E	B91002F	B91002G	B91002H	B91002I
B91002J	B91002K	B91002L	B95030A	B95061A	B95061F	B95061G
B95077A	B97103E	B97104G	BA1001A	BA1101B	BC1109A	BC1109C
BC1109D	BC1202A	BC1202F	BC1202G	BE2210A	BE2413A	

"PRAGMA ELABORATE (REPORT)" has been added at appropriate points in order to solve the elaboration problems for:

C83030C C86007A

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For each chapter, a command file was generated that loaded and executed every program.

For a point of contact for technical information about this Ada implementation system, see:

Mr. Knud Joergen Kirkegaard
DDC International A/S
Gl. Lundtoftevej 1B
DK-2800 Lyngby
DENMARK

Telephone: + 45 42 87 11 44
Telefax: + 45 42 87 22 17

For a point of contact for sales information about this Ada implementation system, see:

In the U.S.A.:

Mr. Mike Turner
DDC-I, Inc.
9630 North 25th Avenue
Suite #118
Phoenix, Arizona 85021

Mailing address:
P.O. Box 37767
Phoenix, Arizona 85069-7767
Telephone: 602-944-1883
Telefax: 602-944-3253

In the rest of the world:

Mr. Palle Andersson
DDC International A/S
Gl. Lundtoftevej 1B
DK-2800 LYNGBY

Denmark

Telephone: + 45 42 87 11 44
Telefax: + 45 42 87 22 17

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro90].

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

a) Total Number of Applicable Tests	3820	
b) Total Number of Withdrawn Tests	81	
c) Processed Inapplicable Tests	269	
d) Non-Processed I/O Tests	0	
e) Non-Processed Floating-Point Precision Tests	0	
f) Total Number of Inapplicable Tests	269	(c+d+e)
g) Total Number of Tests for ACVC 1.11	4170	(a+b+f)

3.3 TEST EXECUTION

Version 1.11 of the ACVC comprises 4170 tests. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors. The AVF determined that 269 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing. In addition, the modified tests mentioned in section 2.3 were also processed.

A magnetic tape containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The test suite was loaded onto a VAX-8530 from the magnetic tape. The test suite was then downloaded onto a Sun3/60 from the VAX-8530 via Ethernet (using DNICP net software utility); the test suite was then loaded via streamer tape to the ICL DRS300.

The tests were compiled, linked and executed on the host/target

computer system, as appropriate. The results were captured on the VAX-8530 computer system using the FTP utility of the TCP/IP data Transfer Services.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options. The options invoked explicitly for validation testing during this test were:

```
-l   -L   -E   -s
```

The options invoked by default for validation testing during this test were:

```
-c
```

Test output, compiler and linker listings, and job logs were captured on magnetic tape and archived at the AVF. Selected listings examined on-site by the validation team were also archived.

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is 126 the value for $MAX_IN_LEN--also listed here. These values are expressed here as Ada string aggregates, where "V" represents the maximum input-line length.

Macro Parameter	Macro Value
$MAX_IN_LEN	126
$BIG_ID1	(1..V-1 => 'A', V => '1')
$BIG_ID2	(1..V-1 => 'A', V => '2')
$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
$BIG_INT_LIT	(1..V-3 => '0') & "298"
$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
$BIG_STRING1	'"' & (1..V/2 => 'A') & '"'
$BIG_STRING2	'"' & (1..V-1-V/2 => 'A') & '1' & '"'
$BLANKS	(1..V-20 => ' ')
$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
$MAX_STRING_LITERAL	'"' & (1..V-2 => 'A') & '"'

The following table contains the values for the remaining macro parameters.

Macro Parameter	Macro Value

```
ACC_SIZE                                : 32
ALIGNMENT                               : 2
COUNT_LAST                              : 2_147_483_647
DEFAULT_MEM_SIZE                        : 16#1_0000_0000#
DEFAULT_STOR_UNIT                       : 16
DEFAULT_SYS_NAME                        : UNIX_V_386
DELTA_DOC                               : 2#1.0#E-31
ENTRY_ADDRESS                           : FCNDECL.ENTRY_ADDRESS
ENTRY_ADDRESS1                          : FCNDECL.ENTRY_ADDRESS1
ENTRY_ADDRESS2                          : FCNDECL.ENTRY_ADDRESS2
FIELD_LAST                              : 65
FILE_TERMINATOR                         : ' '
FIXED_NAME                              : NO_SUCH_FIXED_TYPE
FLOAT_NAME                              : NO_SUCH_FLOAT_TYPE
FORM_STRING                             : ""
FORM_STRING2                            :
   "CANNOT_RESTRICT_FILE_CAPACITY"
GREATER_THAN_DURATION                   : 100000.0
GREATER_THAN_DURATION_BASE_LAST         : 200000.0
GREATER_THAN_FLOAT_BASE_LAST            : 16#1.0#E+32
GREATER_THAN_FLOAT_SAFE_LARGE           : 16#5.FFFF_F0#E+31
GREATER_THAN_SHORT_FLOAT_SAFE_LARGE: 16#5.FFFF_F0#E+31
HIGH_PRIORITY                           : 31
ILLEGAL_EXTERNAL_FILE_NAME1             :
  /usr6/acvc111/list/ctests/ce/name_exceeding_14
ILLEGAL_EXTERNAL_FILE_NAME2             : /../../illegal/nonexistent
INAPPROPRIATE_LINE_LENGTH               : -1
INAPPROPRIATE_PAGE_LENGTH               : -1
INCLUDE_PRAGMA1                         :
   PRAGMA INCLUDE ("A28006D1.TST")
INCLUDE_PRAGMA2                         :
   PRAGMA INCLUDE ("B28006D1.TST")
INTEGER_FIRST                           : -2147483648
INTEGER_LAST                            :  2147483647
INTEGER_LAST_PLUS_1                     :  2147483648
INTERFACE_LANGUAGE                      : C86
LESS_THAN_DURATION                      : -100000.0
LESS_THAN_DURATION_BASE_FIRST           : -200000.0
LINE_TERMINATOR                         : ' '
LOW_PRIORITY                            : 0
MACHINE_CODE_STATEMENT                  :
   MACHINE_INSTRUCTION'(none,m_RETN);
MACHINE_CODE_TYPE                       : MACHINE_STRING
MANTISSA_DOC                            : 31
```

```
MAX_DIGITS                              : 15
MAX_INT                                 :  9223372036854775807
MAX_INT_PLUS_1                          :  9223372036854775808
MIN_INT                                 : -9223372036854775808
NAME                                    : NO_SUCH_TYPE_AVAILABLE
NAME_LIST                               : UNIX_V_386
NAME_SPECIFICATION1                     :
 /usr6/acvc111/list/ctests/ce/X2120A
NAME_SPECIFICATION2                     :
 /usr6/acvc111/list/ctests/ce/X2120B
NAME_SPECIFICATION3                     :
 /usr6/acvc111/list/ctests/ce/X3119A
NEG_BASED_INT                           : 16#F00000000000000E#
NEW_MEM_SIZE                            : 16#100000000#
NEW_STOR_UNIT                           : 16
NEW_SYS_NAME                            : UNIX_V_386
PAGE_TERMINATOR                         : ASCII.FF
RECORD_DEFINITION                               : RECORD OPERAND_KIND :
OPERAND_TYPE; OPCODE : OPCODE_TYPE; END RECORD;
RECORD_NAME                             : MACHINE_INSTRUCTION
TASK_SIZE                               : 32
TASK_STORAGE_SIZE                       : 1024
TICK                                    : 0.000_000_062_5
VARIABLE_ADDRESS                        : FCNDECL.VARIABLE_ADDRESS
VARIABLE_ADDRESS1                       : FCNDECL.VARIABLE_ADDRESS1
VARIABLE_ADDRESS2                       : FCNDECL.VARIABLE_ADDRESS2
YOUR_PRAGMA                             : EXTERNAL_NAME
```

APPENDIX B

COMPILATION SYSTEM OPTIONS

The compiler options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.

QUALIFIER	DESCRIPTION
-a	Auto inline of small local subprograms.
-c <file>	Specifies the configuration file.
-d	Generates debug information.
-E	Generates expanded error messages in the list file.
-l <library>	Specifies program library used.
-L	Generates list file.
-n <checks>	Suppresses the specified run-time checks.
-o <kind>	Performs the specified optimizations.
-s	Inhibits copying Ada source text into the program library.
-x	Creates a cross reference listing.

LINKER OPTIONS

The linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to linker documentation and not to this report.

QUALIFIER	DESCRIPTION
-a <integer>	Main program stack size.
-A <integer>	Library task stack size.
-d	Generates debug information for the DACS 80386 UNIX V Ada debugger.
-e <integer>	Main program segment size
-E <integer>	Library task segment size.
-g <integer>	Tasks default storage size.
-h <integer>	Initial heap size.
-j <file(s)>	Include library archives or object modules in the ld link.
-l <library>	Specifies program sublibrary.
-L	Specifies creation of a log file.
-m <string>	Sign on/off message.
-n	Specifies Ada link only.
-o <file>	Specifies the name of the executable program.
-O <string>	Specifies the ld link options.
-p <integer>	Default task priority.
-P <file>	Specifies template file.
-q	Removes unused code.
-r <recompilation_spec>	Hypothetical recompilation units.
-s	Specifies shared run-time system code should be used.
-T <integer>	Maximum number of tasks.
-v <integer>	Size of reserve stack.
-x	Extracts Ada object modules.
-Z	Use objects in root sublibrary instead of root library archive.
<unit-name>	Specifies the name of the main program to be linked.

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

```
package STANDARD is

type SHORT_INTEGER is range -32_768 .. 32_767;

type INTEGER is range -2_147_483_648 .. 2_147_483_647;

type LONG_INTEGER is range
       -16#8000_0000_0000_0000# .. 16#7FFF_FFFF_FFFF_FFFF#;

type FLOAT is digits 6
     range -16#0.FFFF_FF#E32 .. 16#0.FFFF_FF#E32;

type LONG_FLOAT is digits 15
     range -16#0.FFFF_FFFF_FFFF_F8#E256 ..
   16#0.FFFF_FFFF_FFFF_F8#E256;
type DURATION is delta 2#1.0#E-14 range -131_072.0 .. 131_071.0;

end STANDARD;
```

APPENDIX F
IMPLEMENTATION-DEPENDENT CHARACTERISTICS

This appendix describes the implementation-dependent characteristics of DACS 80386 UNIX V as required in Appendix F of the Ada Reference Manual (ANSI/MIL-STD-1815A).

F.1 Implementation-Defined Pragmas

This section describes all implementation-defined pragmas.

F.1.1 Pragma INTERFACE_SPELLING

This pragma allows an Ada program to call a non-Ada program whose name contains characters that would be an invalid Ada subprogram identifier. It can also be used when subprogram names are case sensitive, e.g. C routines. This pragma must be used in conjunction with pragma INTERFACE, i.e. pragma INTERFACE must be specified for the non-Ada subprogram name prior to using pragma INTERFACE_SPELLING.

The pragma has the format:

pragma INTERFACE_SPELLING (subprogram name, string literal);

where the subprogram name is that of one previously given in pragma INTERFACE and the string literal is the exact spelling of the interfaced subprogram in its native language.

F.1.2 Pragma EXTERNAL_NAME

F.1.2.1 Function

The pragma EXTERNAL_NAME is designed to make permanent Ada objects and subprograms externally available using names supplied by the user.

F.1.2.2 Format

The format of the pragma is:

pragma EXTERNAL_NAME(<ada_entity>,<external name>)

where <ada_entity> should be the name of :

- a permanent object, i.e. an object placed in the permanent pool of the compilation unit - such objects originate in package specifications and bodies only.

- a constant object, i.e. an object placed in the constant pool of the compilation unit - please note that scalar constants are embedded in the code, and composite constants are not always placed in the constant pool, because the constant is not considered constant by the compiler.

- a subprogram name, i.e. a name of a subprogram defined in this compilation unit - please notice that separate subprogram specifications cannot be used, the code for the subprogram MUST be present in the compilation unit code,

and where the <external name> is a string specifying the external name associated the <ada_entity>. The <external names> should be unique. Specifying identical spellings for different <ada_entities> will generate errors at compile and/or link time, and the responsibility for this is left to the user. Also the user should avoid spellings similar to the spellings generated by the compiler, e.g. E_xxxxx_yyyyy, P_xxxxx, C_xxxxx and other internal identifications.

F.1.2.3 Restrictions

Objects that are local variables to subprograms or blocks cannot have external names associated. The entity being made external ("public") MUST be defined in the compilation unit itself. Attempts to name entities from other compilation units will be rejected with a warning.

When an entity is an object the value associated with the symbol will be the relocatable address of the first byte assigned to the object.

F.1.2.4 Example

Consider the following package body fragment:

```
package body example is

   subtype string10 is string(1..10);

   type s is
      record
        len    : integer;
        val    : string10;
      end record;

   global_s    : s;
   const_s     : constant string10 := "1234567890";

   pragma EXTERNAL_NAME(global_s, "GLOBAL_S_OBJECT");
   pragma EXTERNAL_NAME(const_s,  "CONST_S");

   procedure handle(...) is
   ...
   end handle;
```

```
    pragma EXTERNAL_NAME(handle, "HANDLE_PROC");

    ...

end example;
```

The objects GLOBAL_S and CONST_S will have associated the names "GLOBAL_S_OBJECT" and "CONST_S". The procedure HANDLE is now also known as "HANDLE_PROC". It is allowable to assign more than one external name to an Ada entity.

F.1.2.5 Object Layouts

Scalar objects are laid out as described in Chapter 10. For arrays the object is described by the address of the first element; the array constraints cannot be found using the pragma and therefore it is recommended only to use arrays with known constraints. Nondiscriminated records take a consecutive number of bytes, whereas discriminated records may contain pointers to the heap. Such complex objects should be made externally visible, only if the user has thorough knowledge about the layout.

F.1.2.6 Calling Ada Subprograms from C

Using Ada entities from the programming language 'C' can be achieved by using the EXTERNAL_NAME pragma. Special care should be taken in a number of cases. First, Ada subprograms can be called from 'C' only if an Ada main program was used to start the application. This is necessary due to Ada's elaboration requirements.

It should also be noted that the 'C' programming language and Ada have opposite parameter stack order, that is, the parameters for a function call in the 'C' programming language are pushed on the stack in the opposite order in which Ada expects to retrieve them off of the stack. Consequently, the order of arguments within the 'C' programming language routine must be opposite the order of the formal parameters in the Ada subprogram.

All parameters in a function call in the 'C' programming language should be considered to be 'OUT' parameters, that is, if the parameters are changed by the called routine written in Ada, the calling routine, written in 'C', will ignore the changes and discard the information. However, due to differences in accessing the parameters, the corresponding formal parameters in Ada must be declared as 'IN OUT' or just 'OUT'.

There are two methods that an Ada subprogram can return information back to a calling 'C' programming language routine. The first method is by using one of the parameters as a pointer (or access type) to the object to be modified. When the calling routine passes a pointer to the object, the called Ada subprogram will have this declared as an access type to the appropriate object and can use this handle to modify the contents of the object.

The second method is by using an Ada function to return the information to the 'C' programming language routine, as all called routines in 'C' are expected to be functions. When returning a function value, the 'C' programming language as implemented by the UNIX C compiler always

expects the return value to be placed in a specific register, namely %eax. An Ada function, on the other hand, when returning objects other than scalars will return the object by reference and use a more appropriate index register for the address of the object. Consequently, to return information from an Ada function to a 'C' routine, it is necessary for Ada functions to always either return a scalar type, or an access type that is converted using UNCHECKED_CONVERSION to a scalar type before returning to the 'C' programming language routine.

F.1.2.7 Parameter Passing in Calling Ada Subprograms

The following section describes briefly the fundamentals regarding parameter passing in connection with Ada subprograms. For more detail, refer to Chapter 10.

Scalar objects are always passed by value. For OUT or IN OUT scalars, code is generated to move the modified scalar to its destination. In this case the stack space for parameters is not removed by the procedure itself, but by the caller. Composite objects are passed by reference. Records are passed via the address of the first byte of the record. Constrained arrays are passed via the address of the first byte (plus a bit offset when a packed array). Unconstrained arrays are passed as constrained arrays plus a pointer to the constraints for each index in the array. These constraints consist of lower and upper bounds, plus the size in words or bits of each element depending if the value is positive or negative respectively. The user should study an appropriate disassembler listing to thoroughly understand the compiler calling conventions.

A function (which can only have IN parameters) returns its result in register(s). Scalar results are registers/float registers only; composite results leave an address in some registers and the rest, if any, are placed on the stack top. The stack still contains the parameters in this case (since the function result is likely to be on the stack), so the caller must restore the stack pointer to a suitable value, when the function call is dealt with. Again, disassemblies may guide the user to see how a particular function call is to be handled.

Thus it is recommended to use the external name pragma for subprograms only if the user has a thorough understanding of the calling conventions.

F.1.3 Examples

This example illustrates how to write routines in C that can call Ada subprograms. It demonstrates the passing several different types of objects from C routines to a Ada routines.

First the Ada program. Notice that it must contain an Ada main program.

```
with text_io; use text_io;
procedure C_routine_interface is

package int_io is new text_io.integer_io(integer);
use int_io;
```

```
type record_type is
  record
      day   : integer range 1..31;
      month : integer range 1..12;
      year  : integer range 1900..2030;
end record;

type array_type is array (1..10) of record_type;

type test_access is access record_type;

procedure ada_procedure_scalar (i : in out integer;
                                j : in out integer ) is

begin
    put_line("Ada procedure with scalar parameters");
    put("Value from Ada of i is :");
    put(integer'image(i));
    new_line;
    put("Value from Ada of j is :");
    put(integer'image(j));
    new_line;
end ada_procedure_scalar;

pragma external_name(ada_procedure_scalar,
                     "ada_proc_scalar");

procedure ada_procedure_string (s : in out string ) is


begin
    put_line("Ada procedure with a string parameter");
    put("Value from Ada of s is :"); new_line; put(s);
    new_line;
end ada_procedure_string;

pragma external_name(ada_procedure_string,
                     "ada_proc_string");

procedure ada_procedure_composite
                       (arr : in out array_type ) is

begin
    put_line("Ada procedure with a composite parameter");
    put("Value from Ada of arr(4).day is :");
    put(integer'image(arr(4).day));
    new_line;
end ada_procedure_composite;
```

```
pragma external_name(ada_procedure_composite,
                     "ada_proc_composite");

procedure ada_procedure_access
                      (acc : in out test_access ) is

begin
    put_line("Ada procedure with a access parameter");
    put("Value from Ada of dereferenced " &
                                    "day field is :");
    put(integer'image(acc.day));
    new_line;
    put("Value from Ada of dereferenced " &
                                    "month field is :");
    put(integer'image(acc.month));
    new_line;
    put("Value from Ada of dereferenced " &
                                    "year field is :");
    put(integer'image(acc.year));
    new_line;
    acc.month := 6;
    put("Value from Ada of modified dereferenced " &
                                    "day field is :");
    put(integer'image(acc.day));
    new_line;
    put("Value from Ada of modified dereferenced " &
                                    "month field is :");
    put(integer'image(acc.month));
    new_line;
    put("Value from Ada of modified dereferenced " &
                                    "year field is :");
    put(integer'image(acc.year));
    new_line;
end ada_procedure_access;

pragma external_name(ada_procedure_access,
                     "ada_proc_access");

function ada_function_scalar return integer is

i : integer := 7;
```

```
    begin
        put_line("Ada function returning a scalar value");
        put("Value from Ada of i is :");
        put(integer'image(i));
        new_line;
        return(i);
    end ada_function_scalar;

    pragma external_name(ada_function_scalar,
                         "ada_func_scalar");

    procedure c_interface;
    pragma interface(C86, c_interface);

begin
    c_interface;
end C_routine_interface;
```

The Ada compilation unit can be compiled as normal, no special options are necessary as shown below:

$ ada c_routine.ada

The listing of the corresponding C routines:

```
void c_interface()

{
    /* Calling an Ada procedure with scalar type */
    {

    /* The simplest of parameters, the passing method is */
    /* straight forward except that parameters are       */
    /* evaluated in reverse order                        */
    /* The two scalar parameters to be used for passing  */
    int     i = 10;
    int     j = 20;

        printf( "\n\nDemonstration of ada_proc_scalar\n\n" );

        printf( "Value from C of i is %d, and j is %d\n", i, j
);

        ada_proc_scalar( j, i );
    }
```

```
    /* Calling an Ada procedure with a string type */
    {
    /* It should be noted that strings as defined from   */
    /* Ada are considered to be constrained arrays.      */
    /* Consequently, an index descriptor needs to the    */
    /* built and passed as well.  The parameters are:    */
    /* 1) the offset in bits from the start of the       */
    /*    'array' to the first element (in this case 0), */
    /* 2) a starting address for the array,              */
    /* 3) the address of the index constraint            */
    /*    descriptor.                                    */
    /* however, the parameters as pushed on the stack by */
    /* C are in reverse order.                           */

    struct index_descriptor
    {
        int     lower_bound;
        int     upper_bound;
        int     element_size;
    } index;

    char    *s =
    "Now is the time for all good persons (non-sexist) to aid
their parties";

        printf( "\n\nDemonstration of ada_proc_string\n\n" );

        /* First,build the index descriptor */
        index.lower_bound = 1;
        index.upper_bound = strlen(s);

        index.element_size = -8; /* size in bits,       */
                                 /* negative because    */
                                 /* array is packed     */

        printf( "Value from C of string s is\n%s\n", s );

        ada_proc_string( &index, s, 0 );
    }


    /* Calling an Ada procedure with composite types     */
    /* (records and arrays)                              */
    {
    /* Ada declaration of a record type used for         */
    /* demonstration                                     */
    /*                                                   */
    /*  type record_type is                              */
    /*    record                                         */
```

```
/*        day    : integer range 1..31;                */
/*        month  : integer range 1..12;                */
/*        year   : integer range 1970..2030;           */
/*  end record;                                        */
/*                                                     */
/* The corresponding C declaration of the same         */
/* record                                              */
struct record_type
{

    int day;
    int month;
    int year;
};

/*                                                     */
/* Ada declaration of a array type used for            */
/* demonstration                                       */
/*                                                     */
/*    type array_type is                               */
/*                     array (1..10) of record_type    */
/*                                                     */
/*                                                     */
/* The corresponding C declaration of the same         */
/* record                                              */

struct record_type      day_date[10];

                /* initialization of day_date */
day_date[0].day =  1; day_date[0].month = 11;
    day_date[0].year = 1954;
day_date[1].day =  2; day_date[1].month = 12;
    day_date[1].year = 1955;
day_date[2].day =  3; day_date[2].month = 13;
    day_date[2].year = 1956;
day_date[3].day =  4; day_date[3].month = 14;
    day_date[3].year = 1957;
day_date[4].day =  5; day_date[4].month = 15;
    day_date[4].year = 1958;
day_date[5].day =  6; day_date[5].month = 16;
    day_date[5].year = 1959;
day_date[6].day =  7; day_date[6].month = 17;
    day_date[6].year = 1960;
day_date[7].day =  8; day_date[7].month = 18;
    day_date[7].year = 1961;
day_date[8].day =  9; day_date[8].month = 19;
    day_date[8].year = 1962;
day_date[9].day = 10; day_date[9].month = 20;
    day_date[9].year = 1963;
```

```
/* All records are passed by reference, since C also */
/* passes arrays by reference, all that is needed is */
/* to pass the array.  If this had been a record,    */
/* then it would be necessary to pass the address of */
/* the record to Ada                                 */

    printf( "\n\nDemonstration of ");
    printf( "ada_proc_composite\n\n" );

    printf( "Value from C of day_date[3].day is %d\n",
                                    day_date[3].day );

    ada_proc_composite( day_date );
}


/* Calling an Ada procedure with access types */
{
struct record_type
{
    int day;
    int month;
    int year;
} access_test;

access_test.day =  25; access_test.month = 12;
        access_test.year = 2001;

printf( "\n\nDemonstration of ada_proc_access\n\n" );

    printf( "Value from C of access_test.day is %d\n",
                                    access_test.day );
    printf( "Value from C of access_test.month is %d\n",
                                    access_test.month );
    printf( "Value from C of access_test.year is %d\n",
                                    access_test.year );

    ada_proc_access( &access_test );

    printf( "Value from C of modified " );
    printf( "access_test.day is %d\n", access_test.day );
    printf( "Value from C of modified " );
    printf("access_test.month is %d\n", access_test.month );
    printf( "Value from C of modified " );
    printf("access_test.year is %d\n", access_test.year );
}
```

```
    /* Calling an Ada function returning scalars */
    {
    int j;
        printf("\n\nDemonstration of ada_func_scalar\n");

        j = ada_func_scalar();

        printf("Value from C of returned scalar is %d\n",j);
    }
}
```

For this example, consider the C routine to be a file name scalar.c, and compiled with the command line

$ cc -c scalar.c

To link the resulting application, the following command line should be used:

$ al -j scalar.o c_routine_interface

When executed, this program will produce the following output:

```
Ada procedure with scalar parameters
Value from Ada of i is : 10
Value from Ada of j is : 20
Ada procedure with a string parameter
Value from Ada of s is :
Now is the time for all good persons (non-sexist) to aid their
parties
Ada procedure with a composite parameter
Value from Ada of arr(4).day is : 4
Ada procedure with a access parameter
Value from Ada of dereferenced day field is : 25
Value from Ada of dereferenced month field is : 12
Value from Ada of dereferenced year field is : 2001
Value from Ada of modified dereferenced day field is : 25
Value from Ada of modified dereferenced month field is : 6
Value from Ada of modified dereferenced year field is : 2001
Ada function returning a scalar value
Value from Ada of i is : 7


Demonstration of ada_proc_scalar

Value from C of i is 10, and j is 20
```

```
Demonstration of ada_proc_string

Value from C of string s is
Now is the time for all good persons (non-sexist) to aid their
parties


Demonstration of ada_proc_composite

Value from C of day_date[3].day is 4


Demonstration of ada_proc_access

Value from C of access_test.day is 25
Value from C of access_test.month is 12
Value from C of access_test.year is 2001
Value from C of modified access_test.day is 25
Value from C of modified access_test.month is 6
Value from C of modified access_test.year is 2001


Demonstration of ada_func_scalar
Value from C of returned scalar is 7
```

F.1.4 Pragma SHARED_DATA

This pragma is used to specify that the static data declared in the specification of the package will be located in a shared data segment. This segment will be attached to a specific virtual address in programs that has a dependency to the package which allows multiple programs to share the data in the package specification.

The first Ada program using the shared data will create the shared data segment and give it a unique internal shared memory identification. Upon attaching the data at the given virtual address the first Ada program will then perform the elaboration of the shared data.

Subsequent Ada programs using the shared data segment will attach the segment at the given virtual address and detect that it has already been elaborated and avoid re-elaborating the package.

When an Ada program terminates, the shared data segment is detached and the last Ada program to have the shared data segment attached will remove the shared segment when terminating.

When the package specification containing the pragma SHARED_DATA is recompiled, it is assigned a new shared memory identification. Thus, any programs linked with the new version of the package will not share data with programs linked with the previous version of the package. Programs linked with the new version will use another shared data segment.

If any errors occur when creating or attaching a shared segment, PROGRAM_ERROR will be raised (as it occurs during elaboration) and the exit status is set to the error number returned by

the system call that fails. The exit status may be inspected by issuing the shell command 'echo $?' when the program is terminated.

F.1.4.1 Format

The format of the pragma is

```
pragma SHARED_DATA(VIRTUAL_PAGE => <virtual page literal>,
                   PROTECTION   => <protection mask>);
```

where the <virtual page literal> gives the virtual page at which the shared data segment is attached.

Example: VIRTUAL_PAGE => 16#80C0_0#

The shared data segment is attached at the virtual address 16#80C0_0000#

<protection mask> indicates the protection with which the shared data segment is created. The protection may be given for the owner, the group, and others.

Example: PROTECTION => 8#666#

Owner, group and others have read and write privileges.

F.1.4.2 Restrictions

The following restrictions apply to packages containing a pragma SHARED_DATA.

- The virtual address at which the shared segment is attached must be aligned on a 4K byte boundary, therefore the virtual page given in the pragma must also be aligned on a 4K byte boundary.

- Only addresses in the following range may be used:

  16#8000_0000# .. 16#8FFF_0000#

  corresponding to virtual page

  16#8000_0# .. 16#8FFF_0#

- Package specifications containing pragma SHARED_DATA should, for clarity, contain nothing except type declarations and object declarations. Subprograms may be declared, however, the code for the subprograms will not be shared.

- No access types or unconstrained types should be used in the package specification. Allocated objects will be placed in the heap of the Ada program allocating the object and will not be accessible from any other Ada programs.

- No constructs requiring run-time system calls are allowed

  a. task types
  b. discriminant records
  c. allocators

- Regardless of the total size of the objects contained in the specification, the size allocated to a shared data package will be a minimum of 4 Kbytes, and will be increased in 4 Kbyte increments.

- The maximum number of package specifications containing pragma SHARED_DATA allowed to be with'ed by one program is limited to 10.

F.1.4.3 Example

This example illustrates two programs that are sharing a record in the system that contains common information.

```
with System;
with Semaphore;

package shared_record is

    -- pragma SHARED_DATA(VIRTUAL_PAGE => 16#80C0_0#,
                          PROTECTION   => 8#666#)

    data_access : semaphore.semaphore_value ;

    data_block_type is record
        count1 : integer ;
        count2 : integer ;
    end ;

    data : data_block_type ;
       j : integer          : = 1;
end shared_record ;

with shared_record ;
procedure program_1 is

begin

    -- < application specific code >

    wait ( data_access ) ;
    data.count1 := data.count1 + 1 ;
    post ( data_access ) ;
```

```
    -- < application specific code >

end ;

with shared_record ;
procedure program_2 is

begin

    -- < application specific code >

    wait ( data_access ) ;
    data.count1 := data.count1 + 1 ;
    post ( data_access ) ;

    -- < application specific code >

end ;
```

The commands to link these programs would be

```
$ al program_1
$ al program_2
```

When executing, both of these programs would access the same data object that is declared in package shared_record. Furthermore, since the semaphore protecting the data item is in the SHARED_DATA segment, it also becomes a shared object, allowing the two separate programs to maintain controlled access to the data object. If the semaphore was not located in a SHARED_DATA segment, each of the programs would have a local semaphore and there would be no controlled access to the object unless other considerations were made.

F.2 Implementation-Dependent Attributes

No implementation-dependent attributes are defined.

F.3 Package System

The package System for DACS 80386 UNIX V is:

```
PACKAGE system IS

   TYPE    word          IS NEW short_integer;
   TYPE    dword         IS NEW integer;
   TYPE    Qword         IS NEW long_integer;

   TYPE    unsignedword  IS RANGE 0..65535;
```

```
   FOR      unsignedword'SIZE use 16;

   TYPE     unsigneddword IS RANGE 0..16#FFFF_FFFF#;
   FOR      unsigneddword'SIZE use 32;

   TYPE     byte          IS RANGE 0..255;
   FOR      byte'SIZE use 8;

   TYPE     address       IS NEW integer;

   SUBTYPE priority       IS integer RANGE 0..31;

   TYPE     name          IS (UNIX_V_386);

   system_name    : CONSTANT name := UNIX_V_386;
   storage_unit   : CONSTANT      := 16;
   memory_size    : CONSTANT      := 16#1_0000_0000#;
   min_int        : CONSTANT      := -16#8000_0000_0000_0000#;
   max_int        : CONSTANT      := 16#7FFF_FFFF_FFFF_FFFF#;
   max_digits     : CONSTANT      := 15;
   max_mantissa   : CONSTANT      := 31;

   fine_delta     : CONSTANT      := 2#1.0#E-31;
   tick           : CONSTANT      := 0.000_000_062_5;

   TYPE interface_language IS
                    (ASM86,     C86,        C86_REVERSE,
                     ASM_ACF,   C_ACF,      C_REVERSE_ACF,
                     ASM_NOACF, C_NOACF,    C_REVERSE_NOACF);

   TYPE exceptionid  IS record
                          unit_number   : unsigneddword;
                          unique_number : unsigneddword;
                        end record;

   TYPE taskvalue      IS NEW integer;
   TYPE acctaskvalue   IS ACCESS TaskValue;
   TYPE semaphorevalue IS NEW integer;

  TYPE semaphore   IS record
                        counter      : integer;
                        first, last  : taskvalue;
                        SQNext       : semaphorevalue; -- Only used
                                                       in HDS
                          end record;

   initsemaphore : CONSTANT semaphore := semaphore'(1, 0, 0, 0);

end SYSTEM;
```

F.4 Representation Clauses

The representation clauses that are accepted are described below. Note that representation specifications can be given on derived types too.

F.4.1 Pragma PACK

Pragma PACK applied on an array type will pack each array element into the smallest number of bits possible, assuming that the component type is a discrete type other than LONG_INTEGER or a fixed point type. Packing of arrays having other kinds of component types have no effect.

When the smallest number of bits needed to hold any value of a type is calculated, the range of the types is extended to include zero.

Pragma PACK applied on a record type will attempt to pack the components not already covered by a representation clause (perhaps none). This packing will begin with the small scalar components and larger components will follow in the order specified in the record. The packing begins at the first storage unit after the components with representation clauses.

The component types in question are the ones defined above for array types.

F.4.2 Length Clauses

Four kinds of length clauses are accepted.

Size specifications:

The size attribute for a type T is accepted in the following cases:

- If T is a discrete type then the specified size must be greater than or equal to the number of bits needed to represent a value of the type, and less than or equal to 32. Note that when the number of bits needed to hold any value of the type is calculated, the range is extended to include 0 if necessary, i.e. the range 3..4 cannot be represented in 1 bit, but needs 3 bits.

- If T is a fixed point type, then the specified size must be greater than or equal to the smallest number of bits needed to hold any value of the fixed point type, and less than 32 bits. Note that the Reference Manual permits a representation, where the lower bound and the upper bound is not representable in the type. Thus the type

  ```
  type FIX is delta 1.0 range -1.0 .. 7.0;
  ```

  is representable in 3 bits. As for discrete types, the number of bits needed for a fixed point type is calculated using the range of the fixed point type possibly extended to include 0.0.

- If T is a floating point type, an access type or a task type the specified size must be equal to the number of bits used to represent values of the type (floating points: 32 or 64, access types : 32 bits and task types : 32 bits).

- If T is a record type the specified size must be greater than or equal to the minimal number of bits used to represent values of the type per default.

- If T is an array type the size of the array must be static, i.e. known at compile time and the specified size must be equal to the minimal number of bits used to represent values of the type per default.

Furthermore, the size attribute has only effect if the type is part of a composite type.

```
type BYTE is range 0..255;
for BYTE'size use 8;
SIXTEEN : BYTE                  -- one word allocated
EIGHT : array(1.4) of BYTE      -- one byte per element
```

Collection size specifications:

Using the STORAGE_SIZE attribute on an access type will set an upper limit on the total size of objects allocated in the collection allocated for the access type. If further allocation is attempted, the exception STORAGE_ERROR is raised. The specified storage size must be less than or equal to INTEGER'LAST.

Task storage size:

When the STORAGE_SIZE attribute is given on a task type, the task stack area will be of the specified size. There is no upper limit on the given size.

Small specifications:

Any value of the SMALL attribute less than the specified delta for the fixed point type can be given.

F.4.3 Enumeration Representation Clauses

Enumeration representation clauses may specify representations in the range of INTEGER'FIRST+1 .. INTEGER'LAST-1. An enumeration representation clause may be combined with a length clause. If an enumeration representation clause has been given for a type the representational values are considered when the number of bits needed to hold any value of the type is evaluated. Thus the type

```
type ENUM is (A,B,C);
for ENUM use (1,3,5);
```

needs 3 bits not 2 bits to represent any value of the type.

F.4.4 Record Representation Clauses

When component clauses are applied to a record type the following restrictions and interpretations are imposed :

- All values of the component type must be representable within the specified number of bits in the component clause.

- If the component type is either a discrete type, a fixed point type, an array type with a discrete type other than LONG_INTEGER, or a fixed point type as element type, then the component is packed into the specified number of bits (see however the restriction in the paragraph above), and the component may start at any bit boundary.

- If the component type is not one of the types specified in the paragraph above, it must start at a storage unit boundary, a storage unit being 16 bits, and the default size calculated by the compiler must be given as the bit width, i.e. the component must be specified as

  component at N range 0 .. 16 * M-1

  where N specifies the relative storage unit number (0,1,...) from the beginning of the record, and M the required number of storage units (1,2,...).

- The maximum bit width for components of scalar types is 32.

- A record occupies an integral number of storage units (even though a record may have fields that only define an odd number of bytes)

- A record may take up a maximum of 32 Kbits

- If the component type is an array type with a discrete type other than LONG_INTEGER or a fixed point type as element type, the given bit width must be divisible by the length of the array, i.e. each array element will occupy the same number of bits.

If the record type contains components which are not covered by a component clause, they are allocated consecutively after the component with the value. Allocation of a record component without a component clause is always aligned on a storage unit boundary. Holes created because of component clauses are not otherwise utilized by the compiler.

F.4.4.1 Alignment Clauses

Alignment clauses for records are implemented with the following characteristics :

- If the declaration of the record type is done at the outermost level in a library package, any alignment is accepted.

- If the record declaration is done at a given static level (higher than the outermost library level, i.e. the permanent area), only long word alignments are accepted.

F.5 Implementation-Dependent Names for Implementation-Dependent Components

None defined by the compiler.

F.6 Address Clauses

This section describes the implementation of address clauses and what types of entities may have their address specified by the user.

F.6.1 Objects

Address clauses are supported for scalar and composite objects whose size can be determined at compile time. The address value must be static. The given address is the virtual address.

F.7 Unchecked Conversions

Unchecked conversion is only allowed for types where objects have the same "size". The size of an object is interpreted as follows

- for arrays it is the number of storage units occupied by the array elements
- for records it is the size of the fixed part of the record, i.e. excluding any dynamic storage allocated outside the record
- for the other non-structured type, the object size is as described in Chapter 9

For scalar types having a size specification special rules apply. Conversion involving such a type is allowed if the given size matched either the specified size or the object size.

Example

```
type ACC is access INTEGER;
function TO_INT is new UNCHECKED_CONVERSION(ACC,INTEGER);
                                                    -- OK

function TO_ACC is new UNCHECKED_CONVERSION(SHORT_INTEGER,ACC,I);
                                                    -- NOT OK

type UNSIGNED is range 0..65535;
for UNSIGNED'SIZE use 16;

function TO_INT is new UNCHECKED_CONVERSION(UNSIGNED,INTEGER);
                                                    -- OK

function TO_SHORT is new
UNCHECKED_CONVERSION(UNSIGNED,SHORT_INTEGER);
                                                    -- OK
```

End example

F.8 Input/Output Packages

The implementation supports all requirements of the Ada language and the POSIX standard described in the document P1003.5 Draft 4.0/WG15-N45. It is an effective interface to the UNIX file system, and in the case of text I/O, it is also an effective interface to the UNIX standard input, standard output, and standard error streams.

This section describes the functional aspects of the interface to the UNIX file system, including the methods of using the interface to take advantage of the file control facilities provided.

The Ada input-output concept as defined in Chapter 14 of the ARM does not constitute a complete functional specification of the input-output packages. Some aspects of the I/O system are not described at all, with others intentionally left open for implementation. This section describes those sections not covered in the ARM. Please notice that the POSIX standard puts restrictions on some of the aspects not described in Chapter 14 of the ARM.

The UNIX operating system considers all files to be sequences of characters. Files can either be accessed sequentially or randomly. Files are not structured into records, but an access routine can treat a file as a sequence of records if it arranges the record level input-output.

Note that for sequential or text files (Ada files not UNIX external files) RESET to mode OUT_FILE will empty the file. Also, a sequential or text file opened as an OUT_FILE will be emptied.

F.8.1 External Files

An external file is either a UNIX disk file, a UNIX FIFO (named pipe), a UNIX pipe, or any device defined in the UNIX directory. The use of devices such as a tape drive or communication line may require special access permissions or have restrictions. If an inappropriate operation is attempted on a device, the USE_ERROR exception is raised.

External files created within the UNIX file system shall exist after the termination of the program that created it, and will be accessible from other Ada programs. However, pipes and temporary files will not exist after program termination.

Creation of a file with the same name as an existing external file will cause the existing file to be overwritten.

Creation of files with mode IN_FILE will cause USE_ERROR to be raised.

The name parameter to the input-output routines must be a valid UNIX file name. If the name parameter is empty, then a temporary file is created in the /usr/tmp directory. Temporary files are automatically deleted when they are closed

F.8.2 File Management

This section provides useful information for performing file management functions within an Ada program.

The only restrictions in performing Sequential and Direct I/O are:

- The maximum size of an object of ELEMENT_TYPE is 2_147_483_647 bits.

- If the size of an object of ELEMENT_TYPE is variable, the maximum size must be determinable at the point of instantiation from the value of the SIZE attribute.

The NAME parameter

The NAME parameter must be a valid UNIX pathname (unless it is the empty string). If any directory in the pathname is inaccessible, a USE_ERROR or a NAME_ERROR is raised.

The UNIX names "stdin", "stdout", and "stderr" can be used with TEXT_IO.OPEN. No physical opening of the external file is performed and the internal Ada file will be associated with the already open external file. These names have no significance for other I/O packages.

Temporary files (NAME = null string) are created using tmpname(3) and are deleted when CLOSED. Abnormal program termination may leave temporary files in existence. The name function will return the full name of a temporary file when it exists.

The FORM parameter

The Form parameter, as described below, is applicable to DIRECT_IO, SEQUENTIAL_IO and TEXT_IO operations. The value of the Form parameter for Ada I/O shall be a character string. The value of the character string shall be a series of fields separated by commas. Each field shall consist of optional separators, followed by a field name identifier, followed by optional separators, followed by "=>", followed by optional separators, followed by a field value, followed by optional separators. The allowed values for the field names and the corresponding field values are described below. All field names and field values are case-insensitive.

The following BNF describes the syntax of the FORM parameter:

```
form              ::= [field {, field}*]

fields            ::= rights | append | blocking |
                      terminal_input | fifo |
                      posix_file_descriptor

rights            ::= OWNER | GROUP | WORLD =>
                      access {,access_underscor}

access            ::= READ | WRITE | EXECUTE | NONE

access_underscor  ::= _READ | _WRITE | _EXECUTE | _NONE

append            ::= APPEND => YES | NO

blocking          ::= BLOCKING => TASKS | PROGRAM

terminal_input    ::= TERMINAL_INPUT => LINES | CHARACTERS
```

fifo ::= FIFO => YES | NO

posix_file_descriptor ::= POSIX_FILE_DESCRIPTOR => 2

The FORM parameter is used to control the following :

- File ownership:

  Access rights to a file is controlled by the following field names "OWNER", "GROUP" and "WORLD". The field values are "READ", "WRITE", "EXECUTE" and "NONE" or any combination of the previously listed values separated by underscores. The access rights field names are applicable to TEXT_IO, DIRECT_IO and SEQUENTIAL_IO. The default value is OWNER => READ_WRITE, GROUP => READ_WRITE and WORLD => READ_WRITE. The actual access rights on a created file will be the default value subtracted the value of the environment variable **umask**.

  Example

  To make a file readable and writable by the owner only, the Form parameter should look something like this:

```
"Owner =>read_write, World=> none, Group=>none"
```

  If one or more of the field names are missing the default value is used (Owner=>read_write, Group=>read_write, World=>read_write). The permission field is evaluated in left-to-right order. An ambiguity may arise with a Form parameter of the following:

```
"Owner=>Read_Execute_None_Write_Read"
```

  In this instance, using the left-to-right evaluation order, the "None" field will essentially reset the permissions to none and this example would have the access rights WRITE and READ.

- Appending to a file:

  Appending to a file is achieved by using the field name "APPEND" and one of the two field values "YES" or "NO". The default value is "NO". "Append" is allowed with both TEXT_IO and SEQUENTIAL_IO. The effect of appending to a file is that all output to that file is written to the end of the named external file. This field may only be used with the "OPEN" operation, using the field name "APPEND" in connection with a "CREATE" operation shall raise USE_ERROR. Furthermore, a USE_ERROR is raised if the specified file is a terminal device or another device.

  Example

  To append to a file, one would write:

```
"Append => Yes"
```

- Blocking vs. non-blocking I/O:

  The blocking field name is "Blocking" and the field values are "TASKS" and "PROGRAM". The default value is "PROGRAM". "Blocking=>Tasks" causes the calling task, but no others, to wait for the completion of an I/O operation. "Blocking=>program" causes the all tasks within the program to wait for the completion of the I/O operation. The blocking mechanism is applicable to TEXT_IO, DIRECT_IO and SEQUENTIAL_IO. UNIX does not allow the support of "BLOCKING=>TASKS" currently.

- How characters are read from the keyboard:

  The field name is "TERMINAL_INPUT" and the field value is either "LINES" or "CHARACTERS". The effect of the field value "Terminal_input => Characters" is that characters are read in a noncanonical fashion with Minimum_count=1, meaning one character at a time and Time=0.0 corresponding to that the read operation is not satisfied until Minimum_Count characters are received. If the field value "LINES" is used the characters are read one line at a time in canonical mode. The default value is Lines. "TERMINAL_INPUT" has no effect if the specified file is not already open or if the file is not open on a terminal. It is permitted for the same terminal device to be opened for input in both modes as separate Ada file objects. In this case, no user input characters shall be read from the input device without an explicit input operation on one of the file objects. The "TERMINAL_INPUT" mechanism is only applicable to TEXT_IO.

- Creation of FIFO files:

  The field name is "Fifo" and the field value is either "YES" or "NO". "FIFO => YES" means that the file shall be a named FIFO file. The default value is "No".

  For use with TEXT_I/O, the "Fifo" field is only allowed with the Create operation. If used in connection with an open operation an USE_ERROR is raised.

  For SEQUENTIAL_IO, the FIFO mechanism is applicable for both the Create and Open operation.

  In connection with SEQUENTIAL_IO, an additional field name "O_NDELAY" is used. The field values allowed for "O_NDELAY" are "YES" and "NO". Default is "NO". The "O_NDELAY" field name is provided to allow waiting or immediate return. If, for example, the following form parameter is given:

  ```
  "Fifo=>Yes, O_Ndelay=>Yes"
  ```

  then waiting is performed until completion of the operation. The "O_Ndelay" field name only has meaning in connection with the FIFO facility and is otherwise ignored.

- Access to Open POSIX files:

  The field name is "POSIX_File_Descriptor". The field value is the character string "2" which denotes the stderr file. Any other field value will result in USE_ERROR being raised. The Name parameter provides the value which will be returned by subsequent usage of the Name function. The operation does not change the state of the file. During the period that the Ada file is open,

the result of any file operations on the file descriptor are undefined. Note that this is a method to make stderr accessible from an Ada program.

File Access

The following guidelines should be observed when performing file I/O operations:

- At a given instant, any number of files in an Ada program can be associated with corresponding external files.
- When sharing files between programs, it is the responsibility of the programmer to determine the effects of sharing files.
- The RESET and OPEN operations to files with mode OUT_FILE will empty the contents of the file in SEQUENTIAL_IO and TEXT_IO.
- Files can be interchanged between SEQUENTIAL_IO and DIRECT_IO without any special operations if the files are of the same object type.

F.8.3 Buffering

The Ada I/O system provides buffering in addition to the buffering provided by UNIX. The Ada TEXT_IO packages will flush all output to the operating system under the following circumstances:

1. The device is a terminal device and an end of line, end of page, or end of file has occurred.
2. The device is a terminal device and the same Ada program makes an Ada TEXT_IO input request or another file object representing the same device.

F.8.4 Package IO_EXCEPTIONS

The specification of package IO_EXCEPTIONS:

```
package IO_EXCEPTIONS is

   -- The order of the following declarations must NOT be
changed:

   STATUS_ERROR     : exception;
   MODE_ERROR       : exception;
   NAME_ERROR       : exception;
   USE_ERROR        : exception;
   DEVICE_ERROR     : exception;
   END_ERROR        : exception;
   DATA_ERROR       : exception;
   LAYOUT_ERROR     : exception;
```

```
end IO_EXCEPTIONS;
```

F.8.5 Package TEXT_IO

The specification of package TEXT_IO:

```
with BASIC_IO_TYPES;

with IO_EXCEPTIONS;
package TEXT_IO is

   type FILE_TYPE is limited private;

   type FILE_MODE is (IN_FILE, OUT_FILE);

   type  COUNT is range 0 .. INTEGER'LAST;
   subtype POSITIVE_COUNT is COUNT range 1 .. COUNT'LAST;
   UNBOUNDED: constant COUNT:= 0; -- line and page length

   -- max. size of an integer output field 2#....#
   subtype FIELD is INTEGER range 0 .. 65;

   subtype NUMBER_BASE is INTEGER range 2 .. 16;

   type TYPE_SET is (LOWER_CASE, UPPER_CASE);

   -- File Management

   procedure CREATE (FILE      : in out FILE_TYPE;
                     MODE  : in FILE_MODE :=OUT_FILE;
                     NAME  : in STRING    :="";
                     FORM  : in STRING    :=""
                     );

   procedure OPEN (  FILE      : in out FILE_TYPE;
                     MODE  : in FILE_MODE;
                     NAME  : in STRING;
                     FORM  : in STRING    :=""
                     );

   procedure CLOSE (FILE : in out FILE_TYPE)
   procedure DELETE        (FILE : in out FILE_TYPE);
   procedure RESET (FILE : in out FILE_TYPE;  MODE: in FILE_MODE);
   procedure RESET (FILE : in out FILE_TYPE);

   function MODE   (FILE : in FILE_TYPE) return FILE_MODE;
   function NAME   (FILE : in FILE_TYPE) return STRING;
   function FORM   (FILE : in FILE_TYPE) return STRING;
```

```
function IS_OPEN        (FILE : in FILE_TYPE return BOOLEAN;

 -- control of default input and output files

procedure SET_INPUT     (FILE : in FILE_TYPE);
procedure SET_OUTPUT    (FILE : in FILE_TYPE);

function STANDARD_INPUT return FILE_TYPE;
function STANDARD_OUTPUT    return FILE_TYPE;

function CURRENT_INPUT  return FILE_TYPE;
function CURRENT_OUTPUT return FILE_TYPE;

-- specification of line and page lengths

procedure SET_LINE_LENGTH   (FILE : in FILE_TYPE; TO : in COUNT);
procedure SET_LINE_LENGTH   (TO : in COUNT);

procedure SET_PAGE_LENGTH   (FILE : in FILE_TYPE; TO : in COUNT);
procedure SET_PAGE_LENGTH   (TO : in COUNT);

function  LINE_LENGTH   (FILE : in FILE_TYPE) return COUNT;
function  LINE_LENGTH   return COUNT;

function  PAGE_LENGTH   (FILE : in FILE_TYPE) return COUNT;
function  PAGE_LENGTH   return COUNT;

-- Column, Line, and Page Control

procedure NEW_LINE   (FILE : in FILE_TYPE;
                SPACING : in POSITIVE_COUNT := 1);
procedure NEW_LINE      (SPACING : in POSITIVE_COUNT := 1);

procedure SKIP_LINE     (FILE : in FILE_TYPE;
                SPACING : in POSITIVE_COUNT := 1);
procedure SKIP_LINE     (SPACING : in POSITIVE_COUNT := 1);

function END_OF_LINE    (FILE : in FILE_TYPE) return BOOLEAN;
function END_OF_LINE    return BOOLEAN;

procedure NEW_PAGE      (FILE : in FILE_TYPE);
procedure NEW_PAGE;

procedure SKIP_PAGE     (FILE : in FILE_TYPE);
procedure SKIP_PAGE;

function END_OF_PAGE    (FILE : in FILE_TYPE)return BOOLEAN;
function  END_OF_PAGE                    return BOOLEAN;
```

```
   function END_OF_FILE     (FILE : in FILE_TYPE)return BOOLEAN;
   function  END_OF_FILE                          return BOOLEAN;

   procedure SET_COL        (FILE    :    in    FILE_TYPE;TO    :in
POSITIVE_COUNT);
   procedure SET_COL        (TO : in POSITIVE_COUNT);

   procedure SET_LINE       (FILE    :    in    FILE_TYPE;TO    :in
POSITIVE_COUNT);
   procedure SET_LINE       (TO : in POSITIVE_COUNT);

   function  COL    (FILE : in FILE_TYPE)  return POSITIVE_COUNT;
   function  COL                           return POSITIVE_COUNT;

   function  LINE   (FILE : in FILE_TYPE)  return POSITIVE_COUNT;
   function  LINE                          return POSITIVE_COUNT;

   function  PAGE   (FILE : in FILE_TYPE)  return POSITIVE_COUNT;
   function  PAGE                          return POSITIVE_COUNT;

   -- Character Input-Output

   procedure GET    (FILE : in FILE_TYPE; ITEM : out CHARACTER);
   procedure GET    (ITEM : out CHARACTER);
   procedure PUT    (FILE : in FILE_TYPE; ITEM : in CHARACTER);
   procedure PUT    (ITEM : in CHARACTER);

   -- String Input-Output

   procedure GET    (FILE : in FILE_TYPE; ITEM : out STRING);
   procedure GET    (ITEM : out STRING);
   procedure PUT    (FILE : in FILE_TYPE; ITEM : in STRING);
   procedure PUT    (ITEM : in STRING);

   procedure GET_LINE       (FILE : in FILE_TYPE;
                  ITEM : out STRING;
                  LAST : out NATURAL);

   procedure GET_LINE       (ITEM  :  out  STRING;        LAST  :  out
NATURAL);
   procedure PUT_LINE       (FILE : in FILE_TYPE;  ITEM : in STRING);
   procedure PUT_LINE       (ITEM : in STRING);
   -- Generic Package for Input-Output of Integer Types

  generic
      type NUM is range <>;
package INTEGER_IO is

     DEFAULT_WIDTH : FIELD                    := NUM'WIDTH;
```

```
    DEFAULT_BASE  : NUMBER_BASE    := 10;

    procedure GET  (FILE  : in FILE_TYPE;
                    ITEM  : out NUM;
                    WIDTH : in FIELD := 0);

    procedure GET  (ITEM  : out NUM;
                    WIDTH : in FIELD := 0);

    procedure PUT  (FILE  : in FILE_TYPE;
                    ITEM  : in NUM;
                    WIDTH : in FIELD     := DEFAULT_WIDTH;
                    BASE  : in NUMBER_BASE:= DEFAULT_BASE);

    procedure PUT  (ITEM  : in NUM;
                    WIDTH : in FIELD     := DEFAULT_WIDTH;
                    BASE  : in NUMBER_BASE:= DEFAULT_BASE);

    procedure GET  (FROM  : in STRING;
                    ITEM  : out NUM;
                    LAST  : out POSITIVE);

    procedure PUT  (TO    : out STRING;
                    ITEM  : in NUM;
                    BASE  : in NUMBER_BASE:= DEFAULT_BASE);

  end INTEGER_IO;

   -- Generic Packages for Input-Output of Real Types

generic
   type NUM is digits <>;
package FLOAT_IO; is

   DEFAULT_FORE    : FIELD := 2;
   DEFAULT_AFT     : FIELD := NUM'DIGITS - 1;
   DEFAULT_EXP     : FIELD    :=              3;

   procedure GET  (FILE    : in FILE_TYPE;
                   ITEM    : out NUM;
                   WIDTH   : in FIELD := 0);
   procedure GET  (ITEM    : out NUM;
                   WIDTH   : in FIELD := 0);

   procedure PUT  (FILE    : in FILE_TYPE;
                   ITEM    : in NUM;
                   FORE    : in FIELD     := DEFAULT_FORE;
                   AFT     : in FIELD     := DEFAULT_AFT;
                   EXP     : in FIELD     := DEFAULT_EXP);
```

```
   procedure PUT  (ITEM    : in NUM;
                   FORE    : in FIELD     := DEFAULT_FORE;
                   AFT     : in FIELD     := DEFAULT_AFT;
                   EXP     : in FIELD     := DEFAULT_EXP);

   procedure GET  (FROM    : in STRING;
                   ITEM    : out NUM;
                   LAST    : out POSITIVE);
   procedure PUT  (TO      : out STRING;
                   ITEM    :  in NUM;
                   AFT     :  in FIELD    := DEFAULT_AFT;
                   EXP     :  in FIELD    := DEFAULT_EXP);

end FLOAT_IO;
generic
   type NUM is delta <>;
package FIXED_IO is

   DEFAULT_FORE    : FIELD := NUM'FORE;
   DEFAULT_AFT     : FIELD := NUM'AFT;
   DEFAULT_EXP     : FIELD := 0;

   procedure GET  (FILE    : in FILE_TYPE;
                   ITEM    : out NUM;
                   WIDTH   : in FIELD     := 0);
   procedure GET  (ITEM    : out NUM;
                   WIDTH   : in FIELD     := 0);

   procedure PUT  (FILE    : in FILE_TYPE;
                   ITEM    : in NUM;
                   FORE    : in FIELD     := DEFAULT_FORE;
                   AFT     : in FIELD     := DEFAULT_AFT;
                   EXP     : in FIELD     := DEFAULT_EXP);
   procedure PUT  (ITEM    : in NUM;
                   FORE    : in FIELD     := DEFAULT_FORE;
                   AFT     : in FIELD     := DEFAULT_AFT;
                   EXP     : in FIELD     := DEFAULT_EXP);

   procedure GET  (FROM    : in STRING;
                   ITEM    : out NUM;
                   LAST    : out POSITIVE);

   procedure PUT  (TO      : out STRING;
                   ITEM    :  in NUM;
                   AFT     :  in FIELD    := DEFAULT_AFT;
                   EXP     :  in FIELD    := DEFAULT_EXP);

end FIXED_IO;
```

```
   -- Generic Package for Input-Output of Enumeration Types

generic
   type ENUM is (<>);
package ENUMERATION_IO is

   DEFAULT_WIDTH    : FIELD          := 0;
   DEFAULT_SETTING          : TYPE_SET     := UPPER_CASE;

   procedure GET (FILE      : in FILE_TYPE; ITEM : out ENUM);
   procedure GET (ITEM      : out ENUM);

   procedure PUT (FILE      : FILE_TYPE;
                  ITEM      : in ENUM;

                  WIDTH     : in FIELD       := DEFAULT_WIDTH;
                  SET       : in TYPE_SET    := DEFAULT_SETTING);

   procedure PUT (ITEM      : in ENUM;
                  WIDTH     : in FIELD       := DEFAULT_WIDTH;
                  SET       : in TYPE_SET    := DEFAULT_SETTING);

   procedure GET (FROM      : in  STRING;
                  ITEM      : out ENUM;
                  LAST      : out POSITIVE);

   procedure PUT (TO        : out STRING;
                  ITEM      : in ENUM;
                  SET       : in  TYPE_SET := DEFAULT_SETTING);

end ENUMERATION_IO;

   -- Exceptions

   STATUS_ERROR     : exception renames IO_EXCEPTIONS.STATUS_ERROR;
   MODE_ERROR       : exception renames IO_EXCEPTIONS.MODE_ERROR;
   NAME_ERROR       : exception renames IO_EXCEPTIONS.NAME_ERROR;
   USE_ERROR        : exception renames IO_EXCEPTIONS.USE_ERROR;
   DEVICE_ERROR     : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
   END_ERROR        : exception renames IO_EXCEPTIONS.END_ERROR;
   DATA_ERROR       : exception renames IO_EXCEPTIONS.DATA_ERROR;
   LAYOUT_ERROR     : exception renames IO_EXCEPTIONS.LAYOUT_ERROR;

private

   type FILE_BLOCK_TYPE is new BASIC_IO_TYPES.FILE_TYPE;

   type FILE_OBJECT_TYPE is
      record
```

```
        IS_OPEN     : BOOLEAN          := FALSE;
        FILE_BLOCK  : FILE_BLOCK_TYPE;
      end record;

   type FILE_TYPE is access FILE_OBJECT_TYPE;
end TEXT_IO;
```

F.8.6 Package LOW_LEVEL_IO

```
package LOW_LEVEL_IO is

  -- The DACS 80386 UNIX V Ada Compiler System does not
  -- provide for any low level IO.

end LOW_LEVEL_IO;
```

F.8.7 Package SEQUENTIAL_IO

```
-- Source code for SEQUENTIAL_IO
with BASIC_IO_TYPES;
with IO_EXCEPTIONS;

generic

   type ELEMENT_TYPE is private;

package SEQUENTIAL_IO is

  type FILE_TYPE is limited private;

  type FILE_MODE is (IN_FILE, OUT_FILE);

-- File management

   procedure        CREATE(FILE : in out FILE_TYPE;
         MODE       : in    FILE_MODE       := OUT_FILE;
         NAME       : in    STRING          := "";
         FORM       : in    STRING          := "");

   procedure OPEN (FILE     : in out FILE_TYPE;
         MODE       : in    FILE_MODE;
         NAME       : in    STRING;
         FORM       : in    STRING   := "");

   procedure CLOSE (FILE    : in out FILE_TYPE);
```

```
   procedure DELETE(FILE    : in out FILE_TYPE);

   procedure RESET (FILE    : in out FILE_TYPE; MODE : in FILE_MODE);

   procedure RESET (FILE    : in out FILE_TYPE);

   function MODE   (FILE    : in FILE_TYPE) return FILE_MODE;

   function NAME   (FILE    : in FILE_TYPE) return STRING;

   function FORM   (FILE    : in FILE_TYPE) return STRING;

   function IS_OPEN(FILE    : in FILE_TYPE) return BOOLEAN;

-- input and output operations

   procedure READ (FILE     : in FILE_TYPE;
                   ITEM     : out ELEMENT_TYPE);

   procedure WRITE (FILE    : in FILE_TYPE;
                    ITEM    : in ELEMENT_TYPE);

   function END_OF_FILE
                   (FILE    : in FILE_TYPE) return BOOLEAN;

-- exceptions

   STATUS_ERROR     : exception renames IO_EXCEPTIONS.STATUS_ERROR;
   MODE_ERROR       : exception renames IO_EXCEPTIONS.MODE_ERROR;
   NAME_ERROR       : exception renames IO_EXCEPTIONS.NAME_ERROR;
   USE_ERROR        : exception renames IO_EXCEPTIONS.USE_ERROR;
   DEVICE_ERROR     : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
   END_ERROR        : exception renames IO_EXCEPTIONS.END_ERROR;
   DATA_ERROR       : exception renames IO_EXCEPTIONS.DATA_ERROR;

private

   type FILE_TYPE is new BASIC_IO_TYPES.FILE_TYPE;

end SEQUENTIAL_IO;
```

F.8.8 Package DIRECT_IO

```
with BASIC_IO_TYPES;
with IO_EXCEPTIONS;

generic
```

```
        type ELEMENT_TYPE is private;

package DIRECT_IO is

        type FILE_TYPE is limited private;

        type FILE_MODE is (IN_FILE, INOUT_FILE, OUT_FILE);

        type COUNT is range 0..INTEGER'LAST;

        subtype POSITIVE_COUNT is COUNT range 1..COUNT'LAST;

    -- File management

  procedure CREATE(FILE     : in out FILE_TYPE;
                   MODE     : in     FILE_MODE:= INOUT_FILE;
                   NAME     : in     STRING    := "";
                   FORM     : in     STRING    := "");

   procedure OPEN  (FILE     : in out FILE_TYPE;
                   MODE     : in       FILE_MODE;
                   NAME     : in       STRING;
                   FORM     : in       STRING := "");

   procedure CLOSE (FILE     : in out FILE_TYPE);

   procedure DELETE(FILE     : in out FILE_TYPE);

   procedure RESET (FILE     : in out FILE_TYPE;
                   MODE     : in       FILE_MODE);

   procedure RESET (FILE     : in out FILE_TYPE);

   function MODE   (FILE     : in    FILE_TYPE) return FILE_MODE;

   function NAME   (FILE     : in    FILE_TYPE) return STRING;

   function FORM   (FILE     : in    FILE_TYPE) return STRING;

   function IS_OPEN(FILE     : in    FILE_TYPE) return BOOLEAN;

-- input and output operations

   procedure READ  (FILE     : in    FILE_TYPE;
                    ITEM     : out   ELEMENT_TYPE;
                    FROM     : in    POSITIVE_COUNT);
   procedure READ  (FILE     : in    FILE_TYPE;
                    ITEM     : out   ELEMENT_TYPE);
```

```
   procedure WRITE (FILE    : in     FILE_TYPE;
                    ITEM    : in     ELEMENT_TYPE;
                    TO      : in     POSITIVE_COUNT);
   procedure WRITE (FILE    : in     FILE_TYPE;
                    ITEM    : in     ELEMENT_TYPE);

   procedure SET_INDEX
                   (FILE    : in FILE_TYPE;
                    TO      : in POSITIVE_COUNT);

   function INDEX  (FILE    : in FILE_TYPE) return POSITIVE_COUNT;

   function SIZE   (FILE    : in FILE_TYPE) return COUNT;

   function END_OF_FILE
                   (FILE    : in FILE_TYPE) return BOOLEAN;

-- exceptions

   STATUS_ERROR    : exception renames IO_EXCEPTIONS.STATUS_ERROR;
   MODE_ERROR      : exception renames IO_EXCEPTIONS.MODE_ERROR;
   NAME_ERROR      : exception renames IO_EXCEPTIONS.NAME_ERROR;
   USE_ERROR       : exception renames IO_EXCEPTIONS.USE_ERROR;
   DEVICE_ERROR    : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
   END_ERROR       : exception renames IO_EXCEPTIONS.END_ERROR;
   DATA_ERROR      : exception renames IO_EXCEPTIONS.DATA_ERROR;

private
   type FILE_TYPE is new BASIC_IO_TYPES.FILE_TYPE;

end DIRECT_IO;
```

F.8.9 Package TERMINAL

The specification of package TERMINAL:

```
with COMMON_DEFS;
use  COMMON_DEFS;

package TERMINAL is

   procedure SET_CURSOR(ROW, COL          : in     INTEGER);

   procedure IN_CHARACTER(CH    :     out CHARACTER);

   procedure IN_INTEGER  (I     :     out INTEGER);
```

```
    procedure IN_LINE      (T    :    out TERMINAL_LINE);

    procedure OUT_CHARACTER(CH   : in     CHARACTER);

    procedure OUT_INTEGER  (I    : in     INTEGER);

    procedure OUT_INTEGER_F(I, W           : in     INTEGER);

    procedure OUT_LINE     (L    : in     STRING);

    procedure OUT_STRING   (S    : in     STRING);

    procedure OUT_NL;

    procedure OUT_FF;

    procedure FLUSH;

    procedure OPEN_LOG_FILE(FILE_NAME      : in     STRING);

    procedure CLOSE_LOG_FILE;

end TERMINAL;
```

F.9 Machine Code Insertions

The reader should be familiar with the code generation strategy and the 80386 instruction set to fully benefit from this section.

As described in chapter 13.8 of the ARM it is possible to write procedures containing only code statements using the predefined package MACHINE_CODE. The package MACHINE_CODE defines the type MACHINE_INSTRUCTION which, used as a record aggregate, defines a machine code insertion. The following sections list the type MACHINE_INSTRUCTION and types on which it depends, give the restrictions, and show an example of how to use the package MACHINE_CODE.

F.9.1 Predefined Types for Machine Code Insertions

The following types are defined for use when making machine code insertions (their type declarations are given in the following pages):

type opcode_type
type operand_type
type register_type
type segment_register
type machine_instruction

The type REGISTER_TYPE defines registers and register combinations. The double register combinations (e.g. BX_SI) can be used only as address operands (BX_SI describing [BX+SI]). The registers STi describe registers on the floating stack. (ST is the top of the floating stack).

The type SEGMENT_REGISTER defines the four segment registers that can be used to overwrite default segments in an address operand.

The type MACHINE_INSTRUCTION is a discriminant record type with which every kind of instruction can be described. Symbolic names may be used in the form name'ADDRESS

```
type opcode_type is (
        -- 8086 instructions:
             m_AAA, m_AAD,  m_AAM,     m_AAS,
             m_ADC, m_ADD,  m_AND,
             m_CALL,        m_CALLN,
             m_CBW, m_CLC,  m_CLD,     m_CLI,
             m_CMC, m_CMP,  m_CMPS,    m_CWD,
             m_DAA, m_DAS,
             m_DEC, m_DIV,  m_HLT,
             m_IDIV,        m_IMUL,    m_IN,m_INC,
             m_INT, m_INTO, m_IRET,
             m_JA,  m_JAE,  m_JB,          m_JBE,
             m_JC,  m_JCXZ, m_JE,          m_JG,
             m_JGE, m_JL,   m_JLE,     m_JNA,
             m_JNAE,        m_JNB,     m_JNBE,m_JNC,
             m_JNE, m_JNG,  m_JNGE,    m_JNL,
             m_JNLE,        m_JNO,     m_JNP,m_JNS,
             m_JNZ, m_JO,   m_JP,          m_JPE,
             m_JPO, m_JS,   m_JZ,          m_JMP,
             m_LAHF,        m_LDS,     m_LES,m_LEA,
             m_LOCK,        m_LODS,
             m_LOOP,        m_LOOPE,   m_LOOPNE,m_LOOPNZ,
             m_LOOPZ,       m_MOV,     m_MOVS,m_MUL,
             m_NEG, m_NOP,  m_NOT,     m_OR,
             m_OUT, m_POP,  m_POPF,    m_PUSH,
             m_PUSHF,
             m_RCL, m_RCR,  m_ROL,     m_ROR,
             m_REP, m_REPE, m_REPNE,
             m_RET, m_RETP, m_RETN,    m_RETNP,
             m_SAHF,
             m_SAL, m_SAR,  m_SHL,     m_SHR,
             m_SBB, m_SCAS,
             m_STC, m_STD,  m_STI,     m_STOS,
             m_SUB, m_TEST, m_WAIT,    m_XCHG,
             m_XLAT,        m_XOR,

-- 8087/80187/80287 Floating Point Processor instructions:
             m_FABS,        m_FADD,        m_FADDD,m_FADDP,
```

```
        m_FBLD,         m_FBSTP,        m_FCHS,m_FNCLEX,
        m_FCOM,         m_FCOMD,        m_FCOMP,m_FCOMPD,
        m_FCOMPP,       m_FDECSTP, m_FDIV,m_FDIVD,
        m_FDIVP,        m_FDIVR,        m_FDIVRD,m_FDIVRP,
        m_FFREE,        m_FIADD,        m_FIADDD,m_FICOM,
        m_FICOMD,       m_FICOMP,  m_FICOMPD,m_FIDIV,
        m_FIDIVD,       m_FIDIVR,       m_FIDIVRD,
        m_FILD,         m_FILDD,        m_FILDL,m_FIMUL,
        m_FIMULD,       m_FINCSTP, m_FNINIT,m_FIST,
        m_FISTD,        m_FISTP,        m_FISTPD,m_FISTPL,
        m_FISUB,
        m_FISUBD,       m_FISUBR,       m_FISUBRD,m_FLD,
        m_FLDD,         m_FLDCW,        m_FLDENV,m_FLDLG2,
        m_FLDLN2,       m_FLDL2E,       m_FLDL2T,m_FLDPI,
        m_FLDZ,         m_FLD1,         m_FMUL,m_FMULD,
        m_FMULP,        m_FNOP,         m_FPATAN,m_FPREM,
        m_FPTAN,        m_FRNDINT, m_FRSTOR,m_FSAVE,
        m_FSCALE,       m_FSETPM,  m_FSQRT,
        m_FST, m_FSTD,      m_FSTCW,
        m_FSTENV,       m_FSTP,         m_FSTPD,m_FSTSW,
        m_FSTSWAX,      m_FSUB,         m_FSUBD,m_FSUBP,
        m_FSUBR,        m_FSUBRD,  m_FSUBRP,m_FTST,
        m_FWAIT,        m_FXAM,         m_FXCH,m_FXTRACT,
        m_FYL2X,        m_FYL2XP1, m_F2XM1,

-- 80186/80286/80386 instructions:
-    Notice that some immediate versions of the 8086 instructions
--  only exist on these targets (shifts,rotates,push,imul,...)

        m_BOUND,        m_CLTS,         m_ENTER,m_INS,
        m_LAR, m_LEAVE,     m_LGDT,     m_LIDT,
        m_LSL, m_OUTS, m_POPA,          m_PUSHA,
        m_SGDT,         m_SIDT,

        m_ARPL,         m_LLDT,         m_LMSW,m_LTR,
        m_SLDT,         m_SMSW,         m_STR,m_VERR,
        m_VERW,

 -- the 80386 specific instructions:
        m_SETA,         m_SETAE,        m_SETB,m_SETBE,
        m_SETC,         m_SETE,         m_SETG,m_SETGE,
        m_SETL,         m_SETLE,        m_SETNA,m_SETNAE,
        m_SETNB,        m_SETNBE,       m_SETNC,m_SETNE,
        m_SETNG,        m_SETNGE,       m_SETNL,m_SETNLE,
        m_SETNO,        m_SETNP,        m_SETNS,m_SETNZ,
        m_SETO,         m_SETP,         m_SETPE,m_SETPO,
        m_SETS,         m_SETZ,

        m_BSF, m_BSR,
```

```
            m_BT,  m_BTC,       m_BTR,      m_BTS,
            m_LFS, m_LGS,       m_LSS,
            m_MOVZX,        m_MOVSX,
            m_MOVCR,        m_MOVDB,        m_MOVTR,
            m_SHLD,         m_SHRD,

-- the 80387 specific instructions:
            m_FUCOM,        m_FUCOMP,       m_FUCOMPP,
            m_FPREM1,       m_FSIN,         m_FCOS,m_FSINCOS,

-- byte/word/dword variants (to be used, when not deductible from
--  context):
            m_ADCB,         m_ADCW,         m_ADCD,
            m_ADDB,         m_ADDW,         m_ADDD,
            m_ANDB,         m_ANDW,         m_ANDD,
            m_BTW, m_BTD,
            m_BTCW,         m_BTCD,
            m_BTRW,         m_BTRD,
            m_BTSW,         m_BTSD,
            m_CBWW,         m_CWDE,
            m_CWDW,         m_CDQ,
            m_CMPB,         m_CMPW,         m_CMPD,
            m_CMPSB,        m_CMPSW,        m_CMPSD,
            m_DECB,         m_DECW,         m_DECD,
            m_DIVB,         m_DIVW,         m_DIVD,

            m_IDIVB,        m_IDIVW,        m_IDIVD,
            m_IMULB,        m_IMULW,        m_IMULD,
            m_INCB,         m_INCW,         m_INCD,
            m_INSB,         m_INSW,         m_INSD,
            m_LODSB,        m_LODSW,        m_LODSD,
            m_MOVB,         m_MOVW,         m_MOVD,
            m_MOVSB,        m_MOVSW,        m_MOVSD,
            m_MOVSXB,       m_MOVSXW,
            m_MOVZXB,       m_MOVZXW,
            m_MULB,         m_MULW,         m_MULD,
            m_NEGB,         m_NEGW,         m_NEGD,
            m_NOTB,         m_NOTW,         m_NOTD,
            m_ORB, m_ORW,       m_ORD,
            m_OUTSB,        m_OUTSW,        m_OUTSD,
            m_POPW,         m_POPD,
            m_PUSHW,        m_PUSHD,
            m_RCLB,         m_RCLW,         m_RCLD,
            m_RCRB,         m_RCRW,         m_RCRD,
            m_ROLB,         m_ROLW,         m_ROLD,
            m_RORB,         m_RORW,         m_RORD,
            m_SALB,         m_SALW,         m_SALD,
            m_SARB,         m_SARW,         m_SARD,
            m_SHLB,         m_SHLW,         m_SHLDW,
```

```
            m_SHRB,         m_SHRW,         m_SHRDW,
            m_SBBB,         m_SBBW,         m_SBBD,
            m_SCASB,        m_SCASW,        m_SCASD,
            m_STOSB,        m_STOSW,        m_STOSD,
            m_SUBB,         m_SUBW,         m_SUBD,
            m_TESTB,        m_TESTW,        m_TESTD,
            m_XORB,         m_XORW,         m_XORD,
            m_DATAB,        m_DATAW,        m_DATAD,

-- Special 'instructions':
            m_label,        m_reset,

    -- 8087 temp real load/store_and_pop:
            m_FLDT,         m_FSTPT);

type operand_type is (
   none,                        -- no operands
   immediate,                       -- one immediate operand
   register,                        -- one register operand
   address,                         -- one address operand
   system_address,              -- one 'address operand
   name,                        -- CALL name
   register_immediate,              -- two operands :
                                -- destination is register
                                -- source is immediate
   register_register,               -- two register operands
   register_address,                -- two operands :
                                -- destination is register
                                -- source is address
   address_register,                -- two operands :
                                -- destination is address
                                -- source is register
   register_system_address,                 -- two operands :
                                -- destination is register
                                -- source is 'address
   system_address_register,                 -- two operands :
                                -- destination is 'address
                                -- source is register
   address_immediate,               -- two operands :
                                -- destination is address
                                -- source is immediate
   system_address_immediate,    -- two operands :
                                -- destination is 'address
                                -- source is immediate
   immediate_register,              -- only allowed for OUT
                                -- port is immediate
                                -- source is register
   immediate_immediate,             -- only allowed for ENTER
   register_register_immediate,             -- allowed for IMULimm,
```

```
SHRDimm,
                              -- SHLDimm
   register_address_immediate, -- allowed for IMULimm
   register_system_address_immediate,     -- allowed for IMULimm
   address_register_immediate, -- allowed for SHRDimm,
                              -- SHLDimm
   system_address_register_immediate      -- allowed for SHRDimm,
                              -- SHLDimm
                  );


type register_type is       (AX, CX, DX, BX,-- word registers
                             SP, BP, SI, DI,-- word registers
                             AL, CL, DL, BL,-- byte registers
                             AH, CH, DH, BH,-- byte registers
                             EAX,ECX,EDX,EBX,-- dword registers
                             ESP,EBP,ESI,EDI,-- dword registers

                             ES, CS, SS, DS,-- selector registers
                             FS, GS,        -- selector registers

                             BX_SI, BX_DI,  -- 8086/80186/80286
                             BP_SI, BP_DI,  -- combinations
                             ST, ST1, ST2, ST3,-- floating registers
                                            ST4,ST5, ST6, ST7,-    -
(stack)
                          nil);

-- the extended registers (EAX .. EDI) plus FS and GS are only
-- allowed in 80386 targets

type scale_type is (scale_1, scale_2, scale_4, scale_8);

subtype machine_string is string(1..100);

type machine_instruction (operand_kind : operand_type) is
   record
      opcode  : opcode_type;

      case operand_kind is
         when immediate =>
            immediate1          : integer; -- immediate

         when register =>
            r_register          : register_type;-- source and/or
                                  -- destination

         when address =>
            a_segment           : register_type;-- source and/or
```

```
                              --  destination
            a_address_base     : register_type;
            a_address_index                : register_type;
            a_address_scale                : scale_type;
            a_address_offset               : integer;
         when system_address =>
            sa_address         : system.address;-- destination

         when name =>
            n_string           : machine_string;-- CALL
                               -- destination

         when register_immediate =>
            r_i_register_to                : register_type;-       -
destination
            r_i_immediate      : integer; -- source

         when register_register =>
            r_r_register_to                : register_type;-       -
destination
            r_r_register_from              : register_type;-- source

         when register_address =>
            r_a_register_to                : register_type;-       -
destination
            r_a_segment        : register_type;-- source
            r_a_address_base               : register_type;
            r_a_address_index  : register_type;
            r_a_address_scale              : scale_type;
            r_a_address_offset : integer;

         when address_register =>
            a_r_segment        : register_type;-- destination
            a_r_address_base               : register_type;
            a_r_address_index  : register_type;
            a_r_address_scale              : scale_type;
            a_r_address_offset : integer;
            a_r_register_from              : register_type;-- source

         when register_system_address =>
            r_sa_register_to               : register_type;-       -
destination
            r_sa_address       : system.address;-- source

         when system_address_register =>
            sa_r_address       : system.address;-- destination
            sa_r_reg_from      : register_type;-- source

         when address_immediate =>
```

```
            a_i_segment         : register_type;-- destination
            a_i_address_base              : register_type;
            a_i_address_index             : register_type;
            a_i_address_scale             : scale_type;
            a_i_address_offset  : integer;
            a_i_immediate       : integer; -- source

         when system_address_immediate =>
            sa_i_address        : system.address;-- destination
            sa_i_immediate      : integer; -- source

         when immediate_register =>
            i_r_immediate       : integer; -- destination
            i_r_register        : register_type;-- source

         when immediate_immediate =>
            i_i_immediate1      : integer; -- immediate1
            i_i_immediate2      : integer; -- immediate2

         when register_register_immediate =>
            r_r_i_register1               : register_type;-       -
destination
            r_r_i_register2               : register_type;-- source1
            r_r_i_immediate               : integer;-- source2

         when register_address_immediate =>
            r_a_i_register      : register_type;-- destination
            r_a_i_segment       : register_type;-- source1
            r_a_i_address_base  : register_type;
            r_a_i_address_index           : register_type;
            r_a_i_address_scale           : scale_type;
            r_a_i_address_offset          : integer;
            r_a_i_immediate               : integer;-- source2

         when register_system_address_immediate =>
            r_sa_i_register               : register_type;-       -
destination
            addr10          : system.address;-- source1
            r_sa_i_immediate              : integer;-- source2

         when address_register_immediate =>
            a_r_i_segment       : register_type;-- destination
            a_r_i_address_base  : register_type;
            a_r_i_address_index           : register_type;
            a_r_i_address_scale           : scale_type;
            a_r_i_address_offset          : integer;
            a_r_i_register      : register_type;-- source1
            a_r_i_immediate               : integer;-- source2
```

```
      when system_address_register_immediate =>
         sa_r_i_address      : system.address;-- destination
         sa_r_i_register                : register_type;-- source1
         sa_r_i_immediate               : integer;-- source2

      when others =>
         null;
   end case;
end record;
```

F.9.2 Restrictions

Only procedures, and not functions, may contain machine code insertions. Also procedures that use machine code insertions must be specified with PRAGMA inline.

Symbolic names in the form x'ADDRESS can only be used in the following cases:

1) x is an object of scalar type or access type declared as an object, a formal parameter, or by static renaming.

2) x is an array with static constraints declared as an object (not as a formal parameter or by renaming).

3) x is a record declared as an object (not a formal parameter or by renaming).

All opcodes defined by the type OPCODE_type except the m_CALL can be used.

Two opcodes to handle labels have been defined:

m_label: defines a label. The label number must be in the range 1 <= x <= 25 and is put in the offset field in the first operand of the MACHINE_INSTRUCTION.

m_reset: used to enable use of more than 25 labels. The label number after a m_RESET must be in the range 1 <= x <=25. To avoid errors you must make sure that all used labels have been defined before a reset, since the reset operation clears all used labels.

All floating instructions have at most one operand which can be any of the following:

- a memory address
- a register or an immediate value
- an entry in the floating stack

F.9.3 Examples

The following section contains examples of how to use the machine code insertions and lists the generated code.

F.9.3.1 Example Using Labels

The following assembler code can be described by machine code insertions as shown:

```
     mov     $7,$eax
     mov     $4,$eax
     cmp     $ecx,$eax
     jg      1:
     je      2:
     mov     $eax,$ecx
 1:  add     $eax,$ecx
 2:
```

```
with MACHINE_CODE; use MACHINE_CODE;
package example_MC is

    procedure test_labels;
    pragma inline (test_labels);

end example_MC;

package body example_MC is


procedure test_labels is

begin

  MACHINE_INSTRUCTION'(register_immediate, m_MOV, EAX, 7);
  MACHINE_INSTRUCTION'(register_immediate, m_MOV, ECX, 4);
  MACHINE_INSTRUCTION'(register_register, m_CMP, EAX, ECX);
  MACHINE_INSTRUCTION'(immediate, m_JG, 1);
  MACHINE_INSTRUCTION'(immediate, m_JE, 2);
  MACHINE_INSTRUCTION'(register_register, m_MOV, ECX, EAX);
  MACHINE_INSTRUCTION'(immediate, m_label, 1);
  MACHINE_INSTRUCTION'(register_register, m_ADD, EAX, ECX);
  MACHINE_INSTRUCTION'(immediate, m_label, 2);

end test_labels;

end example_MC;
```

F.9.4 Advanced Topics

This section describes some of the more intricate details of the workings of the machine code insertion facility. Special attention is paid to the way the Ada objects are referenced in the machine code body, and various alternatives are shown.

F.9.4.1 Address Specifications

Package MACHINE_CODE provides two alternative ways of specifying an address for an instruction.

The first way is referred to as SYSTEM_ADDRESS and the parameter associated this one must be specified via OBJECT'ADDRESS in the actual MACHINE_CODE insertion. The second way closely relates to the addressing which the 80386 UNIX machine employs: an address has the general form

[base+index*scale+offset]

The ADDRESS type expects the machine insertion to contain values for ALL these fields. The default value NIL for segment, base, and index may be selected (however, if base is NIL, index should be also). Scale MUST always be specified as scale_1, scale_2, scale_4, or scale_8. The offset value must be in the range $-2^{31}..2^{31}-1$.

F.9.4.2 Referencing Procedure Parameters

The parameters of the procedure that consists of machine code insertions may be referenced by the machine insertions using the SYSTEM_ADDRESS or ADDRESS formats explained above. However, there is a great difference in the way in which they may be specified; whether the procedure is specified as INLINE or not.

INLINE machine insertions can deal with the parameters (and other visible variables) using the SYSTEM_ADDRESS form. This will be dealt with correctly even if the actual values are constants. Using the ADDRESS form in this context will be the user's responsibility since the user obviously attempts to address using register values obtained via other machine insertions. It is in general not possible to load the address of a parameter because an 'address' is a two component structure (selector and offset), and the only instruction to load an immediate address is the LEA, which will only give the offset. If coding requires access to addresses like this, one cannot INLINE expand the machine insertions. Care should be taken with references to objects outside the current block since the code generator in order to calculate the proper frame value (using the display in each frame) will apply extra registers. The parameter addresses will, however, be calculated at the entry to the INLINE expanded routine to minimize this problem. INLINE expanded routines should NOT employ any RET instructions. Pure procedure machine insertions need to know the layout of the parameters presented to, in this case, the called procedure. In particular, careful knowledge about the way parameters are passed is required to achieve a successful machine procedure. Again there are two alternatives:

The first assumes that the user takes over the responsibility for parameter addressing. With this method, the SYSTEM_ADDRESS format does not make sense (since it expects a procedural setup that is not set up in a machine procedure). The user must code the exit from the procedure and is also responsible for taking off parameters if so is required. The rules of Ada procedure calls must be followed. The calling conventions are summarized below.

The second alternative assumes that a specific abstract A-code insertion is present in the beginning and end of the machine procedure. Abstract A-code insertions are not generally available to an Ada user since they require extensive knowledge about the compiler intermediate text called abstract A-code. Thus, they will not be explained further here except for the use below.

These insertions enable the user to setup the procedural frame as expected by Ada and then allow the form SYSTEM_ADDRESS into accesses to parameters and variables. Again it is required to know the calling conventions to some extent; mainly to the extent that the access method for variables is clear. A record is, for example, transferred via its address, so access to record fields must first employ an LES-instruction and then use ADDRESS form using the read registers.

The insertions to apply in the beginning are:

```
pragma abstract_acode_insertions(true);
 aa_instr'(aa_Create_Block,x,y,0,0,0);
 aa_instr'(aa_End_of_declpart,0,0,0,0,0);
pragma abstract_acode_insertions(false);
```

and at the end:

```
pragma abstract_acode_insertions(true);
 aa_instr'(aa_Exit_subprgrm,x,0,x,nil_arg,nil_arg); -- (1)
 aa_instr'(aa_Set_block_level,y-1,0,0,0,0);
pragma abstract_acode_insertions(false);
```

where the x value represents the number of words taken by the parameters, and y is the lexical block level of the machine procedure. However, if the procedure should leave the parameters on the stack (scalar IN OUT or OUT parameters), then the Exit_subprgrm insertion should read:

```
aa_instr'(aa_Exit_subprgrm,0,0,0,nil_arg,nil_arg); -- (2)
```

In this case, the caller moves the updated scalar values from the stack to their destinations after the call.

The NIL_ARG should be defined as :

```
nil_arg : constant := -32768;
```

WARNING: When using the AA_INSTR insertions, great care must be taken to assure that the x and y values are specified correctly. Failure to do this may lead to unpredictable crashes in compiler pass8.

F.9.4.3 Parameter Transfer

It may be a problem to figure out the correct number of words which the parameters take up on the stack (the x value). The following is a short description of the transfer method:

INTEGER types take up at least 1 storage unit. 32 bit integer types take up 2 words, and 64 bit integer types take up 4 words. 16 bit integer types take up 2 words; the low word being the value, and the high word being an alignment word. TASKs are transferred as INTEGER.

ENUMERATION types take up as 16 bit INTEGER types (see above).

FLOAT types take up 2 words for 32 bit floats and 4 words for 64 bit floats.

ACCESS types are considered an unsigned INTEGER type (32 bit). When 32 bit offset value, the segment value takes up 2 words the, high word being the alignment word. The offset word(s) are the lowest, and the segment word(s) are the highest.

RECORD types are always transferred by address. A record is never a scalar value (so no post-procedure action is carried out when the record parameter is OUT or IN OUT). The representation is as for ACCESS types.

ARRAY values are transferred as one or two ACCESS values. If the array is constrained, only the

array data address is transferred in the same manner as an ACCESS value. If the array is unconstrained below, the data address will be pushed by the address of the constraint. In this case, the two ACCESS
values will NOT have any alignment words.

Packed ARRAY values (e.g. STRING types) are transferred as ARRAY values with the addition of an INTEGER bit offset as the highest word(s):

```
+H: BIT_OFFSET
+L: DATA_ADDRESS
+0: CONSTRAINT_ADDRESS      -- may be missing
```

The values L and H depend on the presence/absence of the constraint address and the sizes of constraint and data addresses.

In the two latter cases, the form parameter'address will always yield the address of the data. If access is required to constraint or bit offset, the instructions must use the ADDRESS form.

F.9.4.4 Example

A small example is shown below:

```
procedure unsigned_add
                  (op1   : in     integer;
                   op2   : in     integer;
                   res   :    out integer);
```

Notice that machine subprograms cannot be functions.

The parameters take up:

op1	: integer	: 2 words
op2	: integer	: 2 words
res	: integer	: 2 words
Total	:	6 words

The body of the procedure might then be the following, assuming that the procedure is defined at outermost package level:

```
procedure unsigned_add
                  (op1 : in     integer;
                   op2 : in     integer;
                   res :    out integer) is
begin
   pragma abstract_acode_insertions(true);
    aa_instr'(aa_Create_Block,3,1,0,0,0);  -- x = 3, y = 1
    aa_instr'(aa_End_of_declpart,0,0,0,0,0);
   pragma abstract_acode_insertions(false);

   machine_instruction'(register_system_address, m_MOV,
```

```
                                EAX, op1'address);
    machine_instruction'(register_system_address, m_ADD,
                                EAX, op2'address);
    machine_instruction'(immediate,               m_JNC,  1);
    machine_instruction'(immediate,               m_INT,  5);
    machine_instruction'(immediate,               m_label,1);
    machine_instruction'(system_address_register, m_MOV,
                                res'address, AX);

    pragma abstract_acode_insertions(true);
     aa_instr'(aa_Exit_subprgrm,0,0,0,nil_arg,nil_arg);-- (2)
     aa_instr'(aa_Set_block_level,0,0,0,0,0);      -- y-1 = 0
    pragma abstract_acode_insertions(false);
end unsigned_add;
```

A routine of this complexity is a candidate for INLINE expansion. In this case, no changes to the above 'machine_instruction' statements are required. Please notice that there is a difference between addressing record fields when the routine is INLINE and when it is not:

```
type rec is
   record
      low       : integer;
      high      : integer;
   end record;

procedure add_32 is
          (op1   : in     integer;
           op2   : in     integer;
           res   :    out rec);
```

The parameters take up 2 + 2 + 2 words = 6 words. The RES parameter will be addressed directly when INLINE expanded, i.e. it is possible to write:

```
machine_instruction'(system_address_register, m_MOV,
                     res'address, EAX);
```

This would, in the not INLINED version, be the same as updating that place on the stack where the address of RES is placed. In this case, the insertion must read:

```
machine_instruction'(register_system_address, m_LES,
                     ESI, res'address);
   -- LES ESI,[BP+...]
machine_instruction'(address_register, m_MOV,
                     ESI, nil, scale_1, 0, EAX);
   -- MOV [SI+0],EAX
```

As may be seen, great care must be taken to ensure correct machine code insertions. A help could be to first write the routine in Ada, then disassemble to see the involved addressings, and finally write the machine procedure using the collected knowledge.

Please notice that INLINED machine insertions also generate code for the procedure itself. This code

will be removed when the /NOCHECK qualifier is applied to the compilation. Also not INLINED procedures using the AA_INSTR insertion, which is explained above, will automatically get a storage_check call (as do all Ada subprograms). On top of that, 8 bytes are set aside in the created frame, which may freely be used by the routine as temporary space. The 8 bytes are located just below the display vector of the frame (from SP and up). The storage_check call will not be generated when the compiler is invoked with /NOCHECK.

The user also has the option NOT to create any blocks at all, but then he should be certain that the return from the routine is made in the proper way (use the RETP instruction (return and pop) or the RET). Again it will help first to do an Ada version and see what the compiler expects to be done.

F.10 Main Program

A valid main program must be either

1. A library procedure without parameters

2. A library function without parameters returning an integer. The integer will be returned to UNIX as the exit status code of the process. There is no checking on the type of the object returned by the main program.

In neither case is the main program checked for the absence of parameters and the execution of a main program with parameters is undefined.